

ARBEITSBERICHTE DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG (INFORMATIK)

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

HERAUSGEBER: H. Billing, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees, H. Niemann,
D. Seitzer, B. Schmidt, H. J. Schneider, H. Wedekind

SPRACHKONSTRUKTE ZUR SPEZIFIKATION UND KORREKTEN IMPLEMENTATION VON SCHUTZPROBLEMEN

Manfred Reitenspieß

DISSERTATION

BAND 16 · NUMMER 8 · ERLANGEN · AUGUST 1983

1983 № 298



ISSN 0344 - 3515

SPRACHKONSTRUKTE ZUR SPEZIFIKATION UND KORREKTEN IMPLEMENTATION VON SCHUTZPROBLEMEN

Der Technischen Fakultät

der Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR - INGENIEUR

vorgelegt

von

Manfred Reitenspieß

Erlangen 1983

SPRACHKONSTRUKTE ZUM SPEZIFIKATION
UND KORREKTEN LÖSUNG
VON SCHWIERIGKEITEN

Die Technische Fakultät
der Universität Erlangen-Nürnberg
hat die Dissertation
DOKTOR - DISSERTATION

Als Dissertation genehmigt von der
Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung: 11.Mai 1983

Tag der Promotion: 12.Juli 1983

Dekan: Prof.Dr.U.Zwicker

Berichterstatter: Prof.Dr.H.J.Schneider

Prof.Dr.F.Hofmann

Zusammenfassung

Schutzprobleme in Rechensystemen werden seit vielen Jahren theoretisch und praktisch untersucht. Durch gesetzliche und organisatorische Maßnahmen sollen Sicherheitsanforderungen in Rechensystemen durchgesetzt werden. Mit Schutzmodellen sollen Schutzprobleme analysiert werden, Schutzmechanismen sollen die Realisierung von Sicherheitsstrategien ermöglichen. Eine Zusammenstellung der verwendeten Modelle und Methoden ergibt, daß Schutzprobleme meist unabhängig vom umgebenden Problembereich untersucht werden. Obwohl Schutzprobleme und Synchronisationsprobleme denselben Ursprung haben, nämlich die Möglichkeit, Ressourcen eines Rechensystems konkurrierend zu benutzen, werden sie getrennt modelliert und untersucht. Auch bei der sprachlichen Behandlung von Schutzproblemen in Spezifikation und Implementation stehen wegen eines fehlenden einheitlichen Modells keine allgemeinen Schutzkonstrukte zur Verfügung.

Die Betriebssystemmaschine ist ein allgemeines Modell für asynchrone Systeme. Darin kann das Prädikat inf definiert werden, das den Informationsflußbegriff formal beschreibt, so daß die Betriebssystemmaschine zu einem Modell für asynchrone, geschützte Systeme erweitert werden kann. Die Simulation bekannter Schutzmodelle mit Hilfe der erweiterten Betriebssystemmaschine zeigt die Allgemeinheit des Modells. Das inf -Prädikat kann als Basis allgemeiner Sprachkonstrukte zur operationellen Spezifikation von Schutzproblemen abstrakter Datentypen verwendet werden. Deshalb wird eine operationelle Spezifikationssprache zur Spezifikation asynchroner abstrakter Datentypen um Schutzkonstrukte erweitert. Die Definition eines abstrakten Typs mit der erweiterten Betriebssystemmaschine erlaubt die Beschreibung und Untersuchung von Schutz- und Synchronisationsproblemen in einem einheitlichen Modell. Der aus der Literatur bekannte Begriff des versteckten Kanals ist damit definierbar. Die Konstrukte lassen sich auch in eine Implementationssprache und in das zugehörige Übersetzungssystem integrieren. Die dynamische Überprüfung von Schutz- und Synchronisationseinschränkungen erfolgt in einem einheitlichen Mecha-

nismus, der in das Laufzeitsystem der Programmiersprache integriert ist. Schließlich werden Verifikationsbedingungen angegeben, die den formalen Korrektheitsbeweis einer Implementation bez. einer Spezifikation erlauben. Es kann eine Konsistenzbedingung für die Spezifikation angegeben werden, die Inkonsistenzen zwischen spezifizierten Informationsflußeinschränkungen und anderen spezifizierten Eigenschaften ausschließt.

Mit Hilfe der vorgeschlagenen Sprachkonstrukte ist die problemnahe Spezifikation und korrekte Implementation von Schutzproblemen, eingebettet in die Beschreibung abstrakter Datentypen, möglich, so daß ein Bezug zum umgebenden Problembereich und zu Synchronisationsproblemen hergestellt werden kann.

Übersicht

Die vorliegende Arbeit besteht im wesentlichen aus zwei Teilen. Kapitel 1 enthält Begriffsbestimmungen und gibt einen Überblick über theoretische und praktische Ansätze zur Behandlung von Schutzproblemen bei der Spezifikation und Implementation von Software-Systemen. In den Kapiteln 2 bis 5 wird eine Methode zur Spezifikation und korrekten Implementation von asynchronen Systemen um Möglichkeiten zur Beschreibung geschützter Systeme erweitert. Kapitel 6 enthält eine Zusammenfassung der wichtigsten Ergebnisse.

Zur Begriffsbestimmung wird im ersten Abschnitt des ersten Kapitels zunächst der Problembereich abgegrenzt. Dadurch entsteht gleichzeitig eine Ordnung der in der Literatur verwendeten Begriffe. Außerdem werden bekannte Modellierungsansätze für Schutz- und Informationsflußprobleme und bekannte Schutzprobleme vorgestellt. Im zweiten Abschnitt werden Mechanismen und Vorgehensweisen in Hardware-Architekturen, Betriebssystemen und Programmiersprachen zur Berücksichtigung von Schutzproblemen erläutert. Der dritte Abschnitt beschreibt Spezifikations- und Verifikationsmöglichkeiten für Schutzprobleme. Schließlich werden im vierten Abschnitt wesentliche Kritikpunkte der vorgestellten Methoden und Maßnahmen zusammengefaßt. Die wichtigsten Gesichtspunkte jedes Abschnitts werden jeweils am Anfang kurz beschrieben, so daß für einen Überblick die weiteren Details übergangen werden können.

Im zweiten Teil der vorliegenden Arbeit, bestehend aus den Kapiteln 2 bis 5, wird eine Methode zur Spezifikation und korrekten Implementation von asynchronen, geschützten Systemen eingeführt. Dazu wird auf der in /KERAS2/ beschriebenen Methode aufgebaut. Zum Verständnis enthält aber jedes Kapitel eine kurze Zusammenfassung der dort entwickelten Konzepte. In Kapitel 2 wird, ausgehend vom Betriebssystemmaschinenkonzept, das inf-Prädikat eingeführt, so daß der Informationsflußbegriff formal definiert werden kann. Auch Zugriffsschutzeinschränkungen sind damit definierbar. In Kapitel 3 wird eine Sprache zur problemnahen formalen Spezifikation asynchroner, geschützter

abstrakter Datentypen vorgestellt. Ein abstrakter Datentyp wird als spezielle Betriebssystemmaschine aufgefaßt, so daß die Schutzkonstrukte auf das inf-Prädikat der Betriebssystemmaschine zurückgeführt werden können. In Kapitel 4 werden zu den Spezifikationskonstrukten analoge Konstrukte in eine Implementationsprache für asynchrone, geschützte abstrakte Datentypen integriert. Schließlich geht Kapitel 5 auf Verifikationsbedingungen ein, die für eine korrekte Implementation der spezifizierten Schutzeinschränkungen erfüllt sein müssen. Alle vier Kapitel beginnen mit einer Beschreibung der aus der Betriebssystemmaschine übernommenen Gedanken und erläutern dann die notwendigen Erweiterungen. Daran schließen sich Beispiele und Bemerkungen an, die die eingeführten Erweiterungen in einen Zusammenhang zu den im ersten Kapitel definierten Begriffen stellen.

Der Anhang enthält neben dem Literaturverzeichnis die Grammatik zur Syntaxbeschreibung der eingeführten Spezifikationsprache aus Kapitel 3 und eine Zusammenstellung der Verifikationsbedingungen aus Kapitel 5. Außerdem soll ein Stichwortverzeichnis das Lesen des Textes erleichtern.

INHALTSVERZEICHNIS

	Zusammenfassung	I
	Übersicht	III
	Inhaltsverzeichnis	V
1	Schutzprobleme und bekannte Lösungsansätze	-1-
1.1	Der Problembereich und damit zusammenhängende Begriffe	-1-
1.1.1	Schutzprobleme und Schutzmaßnahmen am Beispiel einer Softwarefirma	-4-
1.1.2	Einordnung von Schutzproblemen und Schutzmaßnahmen	-7-
	Sicheres(1) System	-11-
	Zuverlässiges (sicheres(3)) System	-13-
	Geschütztes System	-13-
	Verifiziertes System	-14-
1.1.3	Formale Modelle und bekannte Schutzprobleme	-15-
	Zugriffsschutzmodell von Harrison	-15-
	Take-Grant-Modell	-18-
1.1.4	Das Informationsflußmodell von Denning	-21-
1.1.5	Weitere Modelle	-26-
1.2	Implementierte Lösungsansätze für Schutzprobleme	-27-
1.2.1	Zugriffsschutzmechanismen in Rechnerarchitekturen	-31-
	Capability-Mechanismen	-34-
1.2.2	Schutzmaßnahmen in Betriebssystemen	-41-

	Das System ADEPT-50	-42-
	Das System HYDRA	-46-
	Das Betriebssystem PSOS	-51-
1.2.3	Schutzkonzepte in Programmiersprachen	-52-
	Nicht allgemein bekannte programmiersprachliche Schutzkonstrukte	-55-
1.3	Methoden und Werkzeuge zur Spezifikation und Verifikation von Schutzproblemen	-59-
1.3.1	Spezifikationsmethoden für Schutzprobleme	-61-
	HDM	-63-
	FDM	-66-
1.3.2	Verifikation von Schutzzeigenschaften in Programmen	-68-
	Verifikationsmodell mit Informationsflußanalyse	-72-
1.4	Anforderungen	-75-
2	Das Informationsflußprädikat	-77-
2.1	Das Modell der Betriebssystemmaschine	-79-
	Sequentieller Prozeß	-79-
	Die Betriebssystemmaschine	-80-
2.2	Informationsfluß	-82-
2.2.1	Semantische Abfrage bzw. Änderung	-83-
2.2.2	Direkter Informationsfluß	-86-
	Variablenzugriff	-90-
	Berücksichtigung von Informationsquellen und -senken	-91-
2.2.3	Informationsübertragung und allgemeiner Informationsfluß	-92-
	Erweiterte Betriebssystemmaschine	-93-
2.3	Vergleich mit bekannten Modellen	-94-

	Das Verbandsmodell von Denning	-94-
	Das Modell der Schutzmatrix	-95-
	Der Sicherheitsbegriff von Popek	-96-
	Eigenschaften des inf-Prädikats	-96-
3	Schutzkonstrukte einer operationellen Spezifikationsprache	-97-
3.1	Spezifikationsprache einer operationellen Spezifikationsmethode	-99-
3.1.1	Die Spezifikationsprache	-100-
3.1.2	Zusammenhang mit der Betriebssystemmaschine	-104-
3.2	Die Spezifikationsprache SPAGAT	-106-
3.2.1	Prozessidentifikation	-107-
3.2.2	Schutzobjekte	-110-
3.2.3	Das 'INF'-Konstrukt	-111-
3.2.4	Zugriffseinschränkungen	-115-
3.3	Beispiele	-117-
3.3.1	N-Erzeuger-1-Verbraucher-Problem	-117-
3.3.2	Datenschutz	-124-
3.3.3	Das Philosophenproblem	-127-
3.4	Bemerkungen	-131-
3.4.1	Vollständigkeitspostulat und "versteckter Kanal"	-131-
3.4.2	Einordnung	-134-
4	Schutzkonstrukte in Programmiersprachen	-136-
4.1	Grundlagen zur Implementation abstrakter	-139-

	Datentypen	
4.1.1	Implementation und angewandtes Auftreten abstrakter Datentypen	-139-
4.1.2	Die Sprachumgebung	-140-
4.1.3	Semantik	-143-
4.2	Schutzkonstrukte	-143-
4.2.1	Schutzobjekte	-144-
	Änderung von Schutzobjekten	-146-
4.2.2	Zugriffseinschränkungen	-146-
4.2.3	Informationsflußeinschränkungen	-147-
4.3	Übersetzung der vorgestellten Konstrukte und ihre Laufzeitunterstützung	-149-
	Die Erzeugung abstrakter Objekte	-149-
4.3.1	Schutzobjekte	-150-
	Übergabe von Schutzparametern	-151-
4.3.2	Zugriffseinschränkungen	-153-
	Berechnung von 'CHANGESET' und 'DEPEND- SET'	-154-
4.3.3	Dynamische Informationsflußanalyse	-155-
4.4	Beispiele	-159-
	Der abstrakte Typ 'pnezev'	-159-
	Das abstrakte Typ 'pphtisch'	-162-
4.5	VAX-Realisierung	-164-
	Der Präprozessor	-165-
	Der Postprozessor	-167-
	Das Laufzeitsystem	-167-
	Abstrakte Objekte	-167-

5	Verifikation spezifizierter und implementierter Schutzprobleme	-169-
5.1	Konsistenzbedingungen der Spezifikation	-171-
	Spezifikation von Informationsflußeinschränkungen	-173-
5.2	Formale Beziehungen zwischen Spezifikation und Implementation	-174-
	Schutzkomponenten	-174-
	Ausführungsbedingungen	-175-
	Informationsflußeinschränkungen	-177-
5.3	Korrektheit der Implementation	-178-
5.4	Verifikation des Datentyps 'pnezev'	-179-
	Verifikationsbedingungen der Spezifikation	-180-
	Verifikation der Schnittstelle Spezifikation - Implementation	-181-
	Verifikation der Implementation	-183-
6	Ergebnisse, weiterführende Arbeiten	-184-
	Ergebnisse	-184-
	Anwendungen der Ergebnisse	-185-

Theoretische Untersuchungen	-186-
Schlußbemerkung	-187-
Anhang I: Syntax der Spezifikations- sprache	-188-
Anhang II: Zusammenstellung der Verifika- tionsbedingungen	-194-
Verifikationsbedingungen der Spezifika- tion	-194-
Verifikationsbedingungen der Schnitt- stelle Spezifikation - Implementation	-194-
Verifikationsbedingung der Implementation	-195-
Anhang III: Literaturverzeichnis	-196-
Anhang IV: Stichwortverzeichnis	-202-

1 Schutzprobleme und bekannte Lösungsansätze

Zunächst sollen der Problemkreis "Schutz in und mit Rechensystemen" und damit zusammenhängende Begriffe definiert und gegeneinander abgegrenzt werden. Daran schließt sich die Beschreibung bekannter Lösungsansätze zum Spezifizieren und Implementieren von Schutzproblemen an, so daß daraus die Problemstellung der vorliegenden Arbeit entwickelt werden kann.

1.1 Der Problembereich und damit zusammenhängende Begriffe

Durch die Verbreitung des Datenschutzgedankens rückten Probleme, die sich aus der Sammlung und automatischen Verarbeitung von Daten ergeben, in das öffentliche Interesse. Um dem Mißbrauch der gesammelten Daten durch Verarbeitung und Weitergabe zu begegnen, wurde das Datenschutzgesetz in seinen verschiedenen Ausprägungen als Bundes- oder Landesgesetz verabschiedet. Auch die Rechte und Pflichten der Datenschutzbeauftragten sind darin verankert. Mit dem Gesetz wird ein spezieller Bereich der automatischen Datenverarbeitung, die Verarbeitung personenbezogener Daten, geregelt. Zur Durchführung des Gesetzes werden im wesentlichen organisatorische Maßnahmen eingesetzt, obwohl technische Maßnahmen zur Vereinfachung des Problems beitragen könnten. Dazu stehen aber nur ungenügende Hilfsmittel in bekannten Rechensystemen zur Verfügung.

Neben dem öffentlichen Interesse an Schutzmaßnahmen gibt es seit der Einführung konkurrierend genutzter Rechensysteme ein Interesse von Herstellern und Betreibern der Systeme daran, den Zugang zu den Systemen und deren Verwendung einzuschränken und zu kontrollieren. Dabei spielen sowohl Abrechnungsgesichtspunkte eine Rolle als auch Wartungsgesichtspunkte. Benutzer sollen weder unberechtigt noch unter falschem Namen Ressourcen erhalten oder den Betrieb stören können. Dazu werden i.allg. technische Maßnahmen eingesetzt, die aber nur an der Benutzer-

schnittstelle wirken und unabhängig voneinander verschiedene Teilbereiche eines Rechensystems abdecken. So werden z.B. Paßwortmechanismen zur Benutzeridentifikation, wahrscheinlichkeitstheoretische Untersuchungen zum Schutz statistischer Datenbanken und kryptographische Methoden zum Schutz vor Leitungsspionage verwendet.

Schließlich haben die Benutzer eines Rechensystems selbst ein Interesse am Schutz ihrer Programme und Daten. Zum einen sollen Fehler oder absichtliche Störungen anderer Benutzer nicht den eigenen Programmbetrieb stören. Zweitens sollen Fehler des benutzten Rechensystems den eigenen Programmbetrieb nicht beeinflussen. Und drittens können geeignete Schutzmaßnahmen auch Fehlerauswirkungen in eigenen Programmen einschränken. Aufgrund fehlender Hilfsmittel und ineffizienter Implementierungen werden aber Schutzmaßnahmen, soweit sie in Rechensystemen vorhanden sind, oft nicht genutzt.

Bild 1 soll die drei vorgestellten Interessenbereiche veranschaulichen. Den Interessenbereichen wird jeweils ein allgemei-

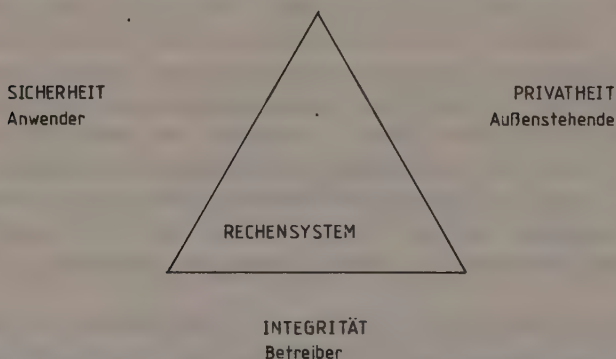


Abb. 1: Interessenbereiche für Schutzmaßnahmen

ner Begriff für die Anforderungen zugeordnet, die die Schutzmaßnahmen erfüllen sollen. Obwohl die Verwendung der Begriffe in der Literatur nicht einheitlich ist, lassen sich Schwerpunkte ausmachen. Mit Privatheit wird das Interesse der (juristischen und natürlichen) Personen bezeichnet, deren personen-

bezogene Daten in Rechensystemen gespeichert und bearbeitet werden. Der Betreiber eines Rechensystems hat Interesse an der Integrität der zur Wartung und zum Betreiben notwendigen logischen und physikalischen Objekte (Verwaltungsinformationen, Geräte). Der Begriff wird deshalb allgemeiner gefaßt als im Datenbankbereich, wo er aber dieselbe Bedeutung im eingeschränkten Sinn hat. Schließlich hat der Anwender eines Rechensystems (Anwendungs-, Systemprogrammierer) Interesse an der Sicherheit seiner logischen Objekte (Programme, Daten), d.h. am Schutz vor Veränderung oder Zerstörung (absichtlich oder unabsichtlich).

Nachdem Schutzmaßnahmen zunächst an Beispielen eingeführt wurden, sollen sie jetzt genauer gefaßt werden. Mit Schutzmaßnahmen werden alle die Maßnahmen in einem Rechensystem bezeichnet, die zum Erreichen von Privatheit, Integrität und Sicherheit der verschiedenen Interessengruppen dienen, die direkt oder indirekt in einem Rechensystem präsent sind. Widersprechen sich verschiedene Maßnahmen hinsichtlich ihrer Wirkung auf Privatheit, Integrität oder Sicherheit, so wird auch dann von Schutzmaßnahmen gesprochen, wenn mindestens eines der Ziele angenähert wird. Es wird nicht zwischen Sicherheits-, Integritäts- und Privatheitsmaßnahmen unterschieden. Analog sollen mit Schutzproblem alle Aspekte einer Rechenanlage bezeichnet werden, die einem der drei Ziele Sicherheit, Integrität oder Privatheit entgegenwirken. Zunächst sollen keine Unterscheidungen hinsichtlich der Art des Schutzproblems getroffen werden.

In diesem Abschnitt werden zunächst im Zusammenhang mit Schutzproblemen stehende Begriffe eingeführt und geordnet. Die weiteren Abschnitte bringen dann bekannte Methoden und Verfahren zur Modellierung und Lösung von Schutzproblemen.



1.1.1 Schutzprobleme und Schutzmaßnahmen am Beispiel einer Softwarefirma

Am Beispiel einer fiktiven Firma soll der Begriff Schutzmaßnahmen verfeinert werden. Die Firma habe hundert Mitarbeiter, die in verschiedenen Filialen in der Bundesrepublik beschäftigt sind. Einige der Filialen haben ein eigenes Rechenzentrum, das sowohl zu Serviceleistungen im Bankgeschäft als auch zur eigenen Softwareentwicklung genutzt wird. Ansonsten gebe es lokale Rechnerkapazität, die zur Softwareentwicklung zur Verfügung steht. Über ein öffentliches Netz seien die Filialen miteinander verbunden.

Zunächst wird eine Gruppe einer Filiale mit ca. 20 Angestellten betrachtet, in der eigenverantwortlich Softwareprojekte durchgeführt werden. Bei der Entwicklung von Programmen muß mit Programmfehlern gerechnet werden, die sich bei Testläufen auf die jeweilige Programmierungsumgebung (Zerstörung von Dateien, Programmen, Geräten) und auf andere Gruppenmitglieder auswirken können. Organisatorische Schutzmaßnahmen sind der exklusive Zugriff zum Entwicklungsrechner und Reduktion der im Rechner befindlichen Daten. Dies ist häufig aus wirtschaftlichen Gründen nicht möglich, so daß technische Maßnahmen wie ein Rechtesystem mit den Rechten "read", "write", "execute" und "delete" auf den Dateien (z.B. VAX unter VMS) vorgezogen werden. Um Querbeziehungen zwischen Teilbereichen eines Projekts zu vermeiden, dürfen Projektmitarbeiter nur über den Projektleiter Informationen untereinander austauschen. Da für ein Projekt gemeinsame Datenbasen existieren, können die Zugriffsbereiche verschiedener Benutzer nicht voneinander getrennt werden. Deshalb sollen technische Maßnahmen wie Rechtevergabe und Führung eines Logbuchs unerlaubte Zugriffe vermeiden. Trotzdem können durch Informationsweitergabe oder auch einfache Kopiervorgänge die Zugriffskontrollen umgangen werden, so daß unter Umständen Informationsflußanalysen durchgeführt werden müssen. Zur Beschränkung der Kommunikation der Projektmitglieder untereinander ist die organisatorische Trennung in disjunkte Teilaufgaben sinnvoll, aber nicht immer durchführbar. Das Isolieren der für jedes Projektmitglied zugänglichen Information ist als organisatorische

Maßnahme nicht sinnvoll, da der Aktualisierungsaufwand sehr groß ist. Dieser Zusammenhang zeigt, daß Schutzmaßnahmen nicht durch äußeren Zwang (Datenschutz), sondern aus technischen Erfordernissen (Verbesserung der Projektrealisierung, Vermeidung von Fehlern bei der Entwicklung) notwendig werden.

Andere Probleme als in einer Projektgruppe ergeben sich bei Zugang zum Rechenzentrum, insbesondere wenn auch Dienstleistungen für Fremdfirmen durchgeführt werden. Der Zugang zum Rechenzentrum selbst und den Datenräumen wird meist durch Ausweise und Schlüssel eingeschränkt. Auch der Zugriff auf Datenträger ist nur einem Teil des Personals erlaubt, zur langfristigen Aufbewahrung stehen spezielle Räume zur Verfügung. Der Rechnerraum selbst wird durch Zutrittskontrollen und spezielle bauliche Maßnahmen geschützt. Der Zugang wird rechnergestützt kontrolliert, z.B. durch Ausweislesegeräte oder die Angabe von Paßwörtern. Durch die Verbesserung von Bild- und Sprachanalysegeräten kann dabei die Vorspiegelung falscher Identitäten weitestgehend ausgeschlossen werden. Es sind also organisatorische und technische Maßnahmen, die die physische und logische Integrität des Rechenzentrums erhalten sollen.

Neben den physischen müssen auch die logischen Zugangsmöglichkeiten zum Rechenzentrum auf Korrektheit und Zulässigkeit überprüft werden. Die Überprüfungen sind i.allg. in die Benutzerschnittstelle des Systems (Kommandointerpreter) integriert. Bei Beginn eines Benutzerauftrags muß der Auftraggeber identifiziert werden, um eine Benutzerumgebung ("domain") aufzubauen, in deren Grenzen sich der Auftrag bewegen darf (aufrufbare Kommandos, zugängliche Dateien, Kontostand, Betriebsmittelverbrauch). Im englischen Sprachraum wird dieser Vorgang mit "authentication" bezeichnet. Meist werden dazu Paßwortverfahren verwendet, wobei die Wahrscheinlichkeit des Mißbrauchs oft von der Paßwortwahl selbst abhängt. Durch Zusatzwissen über den Benutzer und seine Arbeit kann das Paßwort oft erraten werden (Vorname, Projektname). Deshalb werden, analog zur physischen Zugangskontrolle, Mustererkennungsverfahren angewandt. Aber auch die Paßwortverfahren können um Abfragesequenzen erweitert werden, wo Zusatzfragen, deren zugehörige Antworten dem fragen-

den Programm bekannt sind, die Vorspiegelung einer falschen Identität erschweren sollen. Die Zuordnungstabelle zwischen Paßwörtern und Benutzern kann codiert abgespeichert werden, so daß der Zugriff auf die Tabelle keine Rückschlüsse auf die Paßwörter zuläßt.

Analog zu Schutzmaßnahmen einzelner Projektgruppen müssen auch die Benutzer eines Rechenzentrums voreinander geschützt werden, ebenso wie die angebotenen Serviceleistungen vor den Benutzern. Es handelt sich zwar prinzipiell um die gleichen Schutzprobleme wie oben, allerdings muß dabei wegen der geänderten Benutzerstruktur auch mit böswilligen Eingriffen gerechnet werden. Mögliche Gefahrenquellen bestehen im lesenden, ändernden oder zerstörenden Zugriff auf die Daten und Programme des Betriebssystems oder anderer Benutzer. Ziele sind dabei die bevorzugte Bearbeitung von Programmen, Umgehung von Kontroll- oder Buchführungsmaßnahmen und der Informationsgewinn von privaten Dateien anderer Benutzer. Auch Fehler des Betriebssystems können sich auf Benutzerprogramme auswirken und zu Beeinträchtigungen führen. Die Einführung von automatischen Datenverarbeitungssystemen ohne Berücksichtigung der davon betroffenen Mitarbeiter erhöht die Gefahr von Sabotageakten gegen das Betriebssystem oder die Datenbestände anderer Benutzer. Technische Schutzmaßnahmen zur Verringerung oder Beseitigung der genannten Gefahren bestehen in Zugriffskontrollen und der Führung von Logbüchern zur Überwachung der Benutzeraktivitäten. Auch die Verwendung spezieller Programmentwicklungsmethoden wie Verifikation und Informationsflußanalyse von Systemprogrammen zählen dazu. Da Betriebssysteme mit heutigen Mitteln nicht vollständig formal untersucht werden können, wird durch hierarchische Zerlegung des Systems versucht, Betriebssystemteile, die Schutzmaßnahmen enthalten, zu isolieren. Dabei wird angenommen, daß die korrekte Implementation der Schutzmaßnahmen wegen ihrer Verifizierbarkeit mögliche Schutzprobleme reduziert. Organisatorische Maßnahmen wie die zeitliche oder örtliche Trennung der Benutzer sind kaum durchführbar, da der Aufwand zu hoch wäre. Die große Anzahl der Aktivitäten kann nur noch automatisch überprüft werden.

Die Liste der Schutzprobleme lässt sich erweitern, wenn das Rechenzentrum über ein Datenbanksystem verfügt. Sowohl bei der statistischen Auswertung als auch bei gezielten Anfragen können in Verbindung mit Zusatzinformationen Detailinformationen über Personen unzulässigerweise gewonnen werden. Dazu wird der Begriff Inferenzproblem eingeführt, zu dessen Lösung spezielle Gegenmaßnahmen eingesetzt werden, die auf statistischen Modellen beruhen. Ähnlich verhält es sich mit dem Schutz von Informationen, die in Rechnernetzen übermittelt werden. Z.B. könnten die Filialen der fiktiven Firma ein öffentliches Netz zum Aufbau eines Rechnerverbunds benutzen. Um die übermittelten Informationen vor Veränderungen oder Erkennung ("wire tapping") zu schützen, werden kryptographische Methoden verwendet.

Auch für die Einhaltung gesetzlicher Auflagen (Datenschutzgesetz) sind geeignete Schutzmaßnahmen vorzusehen. Mögliche Maßnahmen liegen sowohl im organisatorischen als auch im technischen Bereich. Häufig müssen die Auflagen erst geeignet umgeformt werden, um entsprechende Maßnahmen zu ihrer Lösung ableiten zu können. Z.B. müssen Zugriffseinschränkungen auf eine Datenbank aus dem Datenschutzgesetz abgeleitet werden. Zum Einsatz automatisch unterstützter Schutzmaßnahmen ist die Formulierung der Auflagen in technischen Termen notwendig, aber nicht immer möglich, wenn der Gesetzestext zu allgemein gehalten ist.

1.1.2 Einordnung von Schutzproblemen und Schutzmaßnahmen

Nach der informellen Einführung von Schutzproblemen und möglichen Gegenmaßnahmen sollen diese geordnet und weitere damit zusammenhängende Begriffe eingeführt werden.

Zunächst kann eine Unterscheidung zwischen technischen und organisatorischen Maßnahmen getroffen werden (Bild 2). Als organisatorische Schutzmaßnahmen werden solche bezeichnet, die sich auf die räumliche und zeitliche Anordnung der Datenträger und der zur Verarbeitung notwendigen Anlagen beziehen. Auch

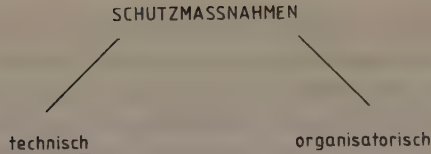


Abb. 2: Technische und organisatorische Schutzmaßnahmen

rechtliche Maßnahmen zur Organisation der Arbeit an und mit Rechenanlagen zählen dazu. Technische Schutzmaßnahmen sind solche, die mit Hilfe technischer Einrichtungen, wozu insbesondere die Rechenanlage selbst zählt, zu ihrer physischen Unversehrtheit und zur logischen Unversehrtheit der damit bearbeiteten Programme und Daten beitragen.

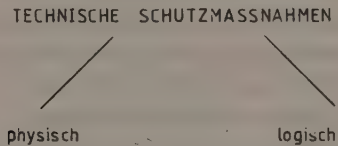


Abb. 3: Physische und logische Schutzmaßnahmen

Obwohl im allgemeinen Sinn des Wortes physische Schutzmaßnahmen die logischen Schutzmaßnahmen enthalten, da Information nicht ohne eine physische Repräsentation existieren kann, sollen physische und logische Schutzmaßnahmen als komplementär betrachtet werden (Abb. 3). Mit physischen Schutzmaßnahmen soll gegen Schutzprobleme vorgegangen werden, die aus Angriffen primär gegen die physische Erscheinungsform eines Rechensystems entstehen. Dazu gehören neben Zerstörungen durch Feuer oder Wasser auch Strahleneinflüsse, die insbesondere in militärisch wichtig eingestuften Anlagen berücksichtigt werden. Demgegenüber wenden sich logische Schutzmaßnahmen gegen solche Schutzprobleme, die aus Angriffen primär gegen die logische Korrektheit eines Rechensystems entstehen. Dazu gehören Penetrationsversuche in das Betriebssystem oder in Bereiche anderer Benutzer, die Manipulation oder Zerstörung wichtiger Daten und Programme oder das Auslesen persönlicher Daten.

Auch Leitungsspionage fällt in diesen Bereich, da neben dem Mithören auch Änderungen der übertragenen Daten möglich sind.

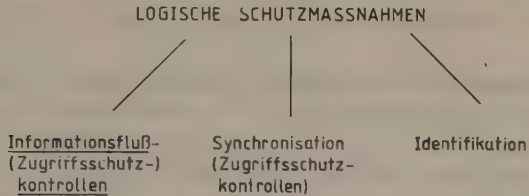


Abb. 4: Informationsflußkontrollen und Synchronisationsmaßnahmen

Informationsflußkontrollen (Abb. 4) sollen allgemein alle Schutzmaßnahmen heißen, die den Informationsfluß in Rechenanlagen kontrollieren. Informationsfluß wird im zweiten Kapitel formal definiert, so daß mögliche Kontrollen ebenfalls exakt beschreibbar werden. Zunächst ist damit die Übertragung von Information mit Hilfe beliebiger unterscheidbarer Objekte eines Rechensystems gemeint. Daneben stehen Synchronisationsmaßnahmen, die den mehrfachen Zugriff auf Informationen kontrollieren. Dazu gehören Koordinierungsmaßnahmen zwischen Prozessen ebenso wie der gegenseitige Ausschuß beim Zugriff auf Informationen. Beide Begriffe sind eng mit Zugriffskontrollen verbunden, da Informationen stets an konkrete Objekte gebunden sind und Informationsfluß durch Zugriffe auf Objekte realisiert wird. Der Vorgang der Identifizierung ("authentication") ist notwendige Voraussetzung für alle Zugriffskontrollen, um Objekte voneinander unterscheiden zu können. Objektidentifizierung wird deshalb auch an den definierten Schnittstellen einer Rechenanlage (Prozeßperipherie, Terminals, Eingabegeräte) betrieben. Dadurch ist sichergestellt, daß externe Informationsquellen oder -senken mit ihren logischen Repräsentanten in der Rechenanlage übereinstimmen. Identifizierungsalgorithmen wurden ursprünglich für Timesharing-Anlagen entwickelt, wo der Benutzername über die Zugriffsmöglichkeiten des damit bezeichneten Benutzers auf das System entscheidet. Deshalb wird Identifizierung als eigenständiges Schutzproblem betrachtet, das mit speziellen Schutzmaßnahmen gelöst wird.

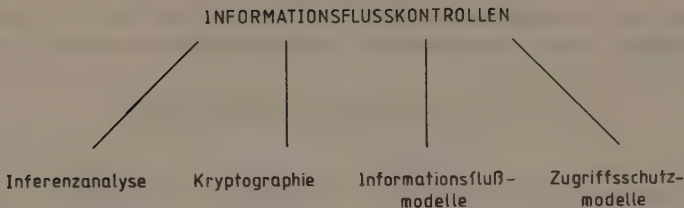


Abb. 5: Identifizierung

Obwohl prinzipiell eine einheitliche Formulierung der mit logischen Schutzmaßnahmen angegangenen Probleme möglich ist, werden die Probleme zur besseren Untersuchung mit verschiedenen Modellen beschrieben. In allen in Bild 5 gezeigten Bereichen geht es um Maßnahmen zur Kontrolle von Informationsflüssen. Dieser einheitliche Aspekt soll bei Entwicklung von sprachlichen Konstrukten ausgenutzt werden. Mit Hilfe von kryptographischen Methoden wird häufig die Konsistenz gespeicherter oder übertragener Informationen sichergestellt. Die Informationen werden dabei verschlüsselt, so daß ohne einen geeigneten Schlüssel gelesene Informationen nicht interpretierbar oder unzulässig geschriebene Informationen als solche erkennbar sind. Die Inferenzanalyse wird speziell beim Schutz statistischer Datenbanken vor dem unerlaubten Zugriff auf Einzelinformationen eingesetzt. Informationsflußmodelle und Zugriffsschutzmodelle sind speziell für Schutzmaßnahmen in Rechensystemen entwickelt und spiegeln die Organisation der darin enthaltenen logischen Elemente (Objekte) wider. Häufig wird dabei zwischen aktiven Einheiten (Subjekte, Prozesse) und passiven Einheiten (Objekte, Dateien, Segmente) unterschieden. Insbesondere können auch Subjekte als Objekte auftreten. Deshalb wird allgemein angenommen, daß die Subjekte eine Teilmenge der Objekte sind. Bei Informationsflußmodellen wird meist der Informationsfluß zwischen verschiedenen Objekten als den Informationsträgern betrachtet.

Im weiteren wird sich die Diskussion auf logische Schutzmaßnahmen beschränken. Im Mittelpunkt werden nicht Modelle zur Analyse von Schutzproblemen stehen, sondern ein allgemeines Modell zur Beschreibung von Schutzproblemen in Rechensystemen.

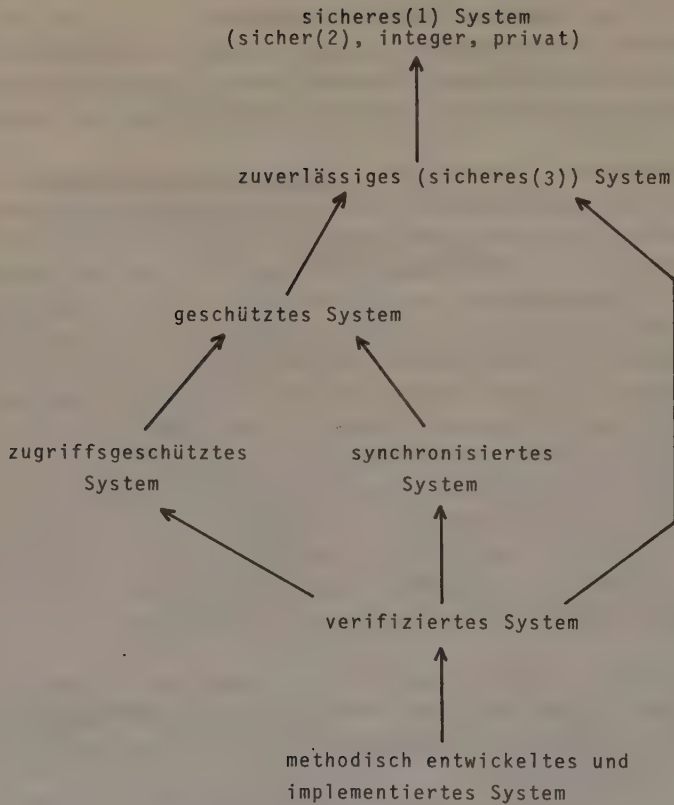
Wegen ihrer Bedeutung werden in Abschnitt 1.1.3 noch die bekanntesten Schutzmodelle anhand einiger charakteristischer Schutzprobleme erläutert. Zunächst sollen aber noch weitere Begriffe im Zusammenhang mit Schutzproblemen eingeführt werden.

Häufig werden die Begriffe Schutzstrategie ("protection policy", "security policy") und Schutzmechanismus ("protection mechanism") verwendet. Während der erste Begriff den Plan zum Erreichen eines vorgegebenen Ziels bedeutet, wird mit Mechanismus ein feststehender Vorgang definiert, der in Abhängigkeit von seiner Umgebung ein berechenbares Ergebnis liefert. In Zusammenhang mit Schutz muß der Strategiebegriff um das zu erreichende Ziel ergänzt werden. Zum Erreichen des vordefinierten Ziels werden vorhandene (Schutz-) Mechanismen verwendet. (Die Verwendung des Begriffs Mechanismus deckt sich nicht mit der des DUDEN-Begriffslexikons. Dort ist mit Mechanismus dessen mechanische Realisierung verbunden. Obwohl Werkzeug dem englischen "mechanism" besser entspricht und auch Algorithmus einen feststehenden, berechenbaren Vorgang bezeichnet, wird der gebräuchliche Begriff Schutzmechanismus beibehalten.)

Schließlich sollen noch zu Schutzmaßnahmen verwandte oder ähnliche Begriffe erläutert werden, um eine Abgrenzung zwischen den in der Literatur nebeneinander gebrauchten Begriffen zu erhalten (Bild 6).

Sicheres(1) System

Der Begriff sicher ("secure", "safe") wird in mehreren Bedeutungen gebraucht, die im Englischen teilweise durch "secure" (zuverlässig, mit Sicherheit) und "safe" (geschützt, in Sicherheit) unterschieden werden. Sicher(1) wird im Sinn von "safe" gebraucht und ist nicht absolut definierbar, sondern abhängig von einem vorgegebenen Sicherheitsbegriff. Dieser kann entweder in einem formalen Modell definiert sein (/HARR76/) oder in einem Gesetz (Datenschutzgesetz). Sicher(2), integer und privat sind Spezialisierungen von sicher(1) und wurden bereits eingeführt. Daraus wird auch ersichtlich, daß ein sicheres(1) System



(a → b: a ist Voraussetzung für b)

Abb. 6: Geschützte Systeme und verwandte Begriffe

nicht nur durch logische Schutzmaßnahmen erreicht werden kann. Das bedeutet, daß sicher(1) nicht geschlossen formal darstellbar ist, sondern aus mehreren zusammenwirkenden Komponenten besteht.

Zuverlässiges (sicheres(3)) System

Sicher(3) entspricht dem englischen "secure" und soll im weiteren durch zuverlässig übersetzt werden. Ein System heißt zuverlässig, wenn es in allen erreichbaren Systemzuständen "angemessen" reagiert. Es müssen also auch Reaktionen auf mögliche unerwartete Ereignisse vorgesehen sein. Die Zuverlässigkeit eines Systems wird wesentlich durch seine Verifikation erhöht, da dadurch zumindest Programmierfehler ausgeschlossen werden können, so daß weniger unerwartete Systemzustände erreichbar sind. Zur Erstellung eines zuverlässigen Systems sind verschiedene Maßnahmen möglich, die jede für sich die Zuverlässigkeit erhöhen. Man geht i.allg. davon aus, daß es kein zu 100 Prozent zuverlässiges System gibt, daß aber das Maximum relativ gut angenähert werden kann. Zu den Maßnahmen zählen:

- spezielle Hardware
- Verifikation
- methodische Entwicklung und Implementation der Software-Komponenten
- Schutzmaßnahmen

Mit Leistungsverweigerung ("denial of service") werden alle Einschränkungen bezeichnet, die die korrekte Ausführung von zulässigen Operationen eines Systems behindern. Dazu gehören physikalische Störungen der Rechenanlage ebenso wie belegte Ressourcen. Durch die unspezifische Verwendung des Begriffs ist er keinem konkreten Schutzproblem zuzuordnen, sondern seine Lösung ist abhängig vom Grad der Zuverlässigkeit.

Geschütztes System

Ein System heißt geschützt, wenn es Schutzmaßnahmen enthält. Schutzmaßnahmen müssen damit nicht immer zu einem sicheren(1) System führen, sondern können auch zur Verbesserung der Zuverlässigkeit dienen. In vielen Aufsätzen wird auf die vereinfachte Fehlersuche bei Verwendung von Kontrollmechanismen für Objektzugriffe hingewiesen. Besonders /LIND76/ stellt einen engen Zusammenhang zwischen Zuverlässigkeit, Schutzmechanismen,

speziellen Systemstrukturen ("principle of least privilege", "small domains") und Entwicklungsprinzipien (formale Verifikation, strukturierte Programmierung) her. Neben Mechanismen zur Kontrolle der Objektzugriffe gehören auch Synchronisationsmechanismen zu einem geschützten System, da sie zur Erhaltung der Konsistenz von Objekten bei mehreren zugreifenden Subjekten notwendig sind. Da dies auch in ungeschützten Systemen gilt, fanden bisher Synchronisationsmaßnahmen bei der Entwicklung von Rechensystemen mehr Aufmerksamkeit als Schutzmaßnahmen.

Verifiziertes System

Die formale Verifikation eines Systems ist zwar nicht hinreichend für ein zuverlässiges System, wegen möglicher Hardware-Fehler, kann aber durch die Formalisierung des Problems und der formalen Überprüfung der Lösung die Zuverlässigkeit erhöhen. Unter die Verifikation fallen auch Informationsflußkontrollen, da diese häufig nicht mit Mechanismen, sondern durch statische formale Untersuchungen realisiert sind. Dynamisch werden Informationsflußkontrollen auf Zugriffskontrollen zurückgeführt, so daß der Begriff "zugriffsgeschützt" beim geschützten System verwendet wird (Bild 6).

Die methodische Entwicklung und Implementation sollte bei jedem System vorausgesetzt werden. Sie ist aber speziell bei zuverlässigen oder sicheren(1) Systemen notwendig, da ohne eine geeignete Strukturierung die notwendigen Schutzmaßnahmen nicht erkannt werden könnten. Dies zeigt sich besonders bei Betriebssystemen, wo häufig nicht beachtete Fälle oder Fehler zur Umgehung der Schutzmechanismen ausgenutzt werden.

1.1.3 Formale Modelle und bekannte Schutzprobleme

Obwohl Schutzmodelle nicht zum eigentlichen Thema der vorliegenden Arbeit gehören, sollen der Vollständigkeit wegen die bekanntesten Modellierungsansätze vorgestellt werden. Um den Problembereich auf Schutzmodelle zu beschränken, werden statistische und kryptographische Modelle nicht berücksichtigt, da diese nicht auf Schutzprobleme eingeschränkt sind. Mit Hilfe der Modelle sollen dann spezielle Schutzprobleme definiert werden.

Zugriffsschutzmodell von Harrison

In /HARR76/ wird (zugriffs-)geschütztes System als 6-Tupel definiert

$$GSI = (S, O, SM, R, SOP, sm_0)$$

S ist die Menge der Subjekte, O die Menge der unterscheidbaren Objekte und $SM = S \times O \times R_{pot}$ (R_{pot} = Potenzmenge) heißt Schutzmatrix. Jedes Element $sm = (s, o, R')$ aus SM bezeichnet die Rechte $R' \subset R$, die Subjekt $s \in S$ bez. Objekt $o \in O$ besitzt. Operationen, die Subjekt s auf Objekt o ausführen will, sind durch das zugeordnete Matrixelement (s, o, R') beschränkt. Die Operationenmenge SOP bezeichnet eine spezielle Menge von Schutzoperationen, die i.allg. besteht aus

- trage $r \in R$ ein in (s, o, R')
- entferne r aus R' , wenn gilt: $(s, o, R') \in SM$
- erzeuge Objekt o
- erzeuge Subjekt s
- zerstöre Subjekt s
- zerstöre Objekt o .

sm_0 ist der Anfangszustand der Schutzmatrix.

Bild 7 zeigt eine Schutzmatrix mit den Benutzern B1 und B2 als Subjekte und den Dateien D1, D2 und D3 als weitere Objekte. Zulässige Operationen sind "benachrichtigen" von Benutzern und "lesen", "schreiben" und "verlängern" von Dateien. Die Rechte entsprechen den gleichnamigen Operationen.

	B ₁	B ₂	D ₁	D ₂	D ₃
B ₁	%	benachrichtigen	lesen schreiben	%	lesen schreiben
B ₂	benachrichtigen	%	lesen	lesen schreiben	verlängern

Abb. 7: Beispiel einer Schutzmatrix

Das Modell ist zur Beschreibung solcher Schutzprobleme geeignet, die sich mit gedächtnislosen Zugriffskontrollen lösen lassen. Gedächtnislos bedeutet, daß eine Zugriffsprüfung nicht von vorausgehenden Zugriffen abhängt. Die Allgemeinheit des Modells erlaubt Entscheidbarkeitsuntersuchungen für einen vorgegebenen Sicherheitsbegriff, der auf der Verteilung von Rechten auf die Schutzmatrix basiert. Informell wird das Sicherheitsproblem ("safety problem") durch die Frage charakterisiert, "ob ein Element der Schutzmatrix ein Recht $r \in R$ erlangen kann". Das Problem ist in dieser Allgemeinheit nicht entscheidbar und auch bei Einschränkungen des Modells, z.B. durch eine endliche Anzahl von Subjekten, nur in polynomialer Zeit entscheidbar (/LIPT78/).

Mit Hilfe des Modells lassen sich zunächst die Begriffe Schutzdomäne ("protection domain") und Prinzip des kleinsten Privilegs ("principle of least privilege") definieren. Die Schutzdomäne $D(s)$ eines Subjekts s ist die Menge der Objekte mit dem dem Subjekt zustehenden Rechten, also die dem Subjekt zugeordnete Zeile der Schutzmatrix:

$$D(s) = \{ (o, R') \in O \times R_{\text{pot}} \mid (s, o, R') \in \text{SM} \}$$

Das Prinzip des kleinsten Privilegs bezieht sich auf die dynamischen Änderungen der Schutzmatrix und verlangt, daß die Schutzdomäne eines Subjekts zu jedem Zeitpunkt möglichst klein ist.

"Möglichst klein" wird nicht im Schutzmodell definiert, sondern hängt vom Differenzierungsgrad der Subjekte und Objekte ab.

Eng mit Schutzdomänen sind Schutzprobleme verbunden, die unter dem Sammelbegriff Trojanisches Pferd ("Trojan horse") bzw. Falltüren ("trapdoors") bekannt sind. Dabei nutzen Subjekte Rechte aus, die ihnen durch seltene Domänenänderungen zugänglich sind. Z.B. könnten Dienstprogramme, die auf Datenbereichen der aufrufenden Benutzer operieren, nicht mit dem eigentlichen Zweck des Aufrufs verbundene Operationen ausführen (Trojanisches Pferd). Durch den Aufruf des Dienstprogramms erhält die dem Subjekt "Dienstprogramm" zugeordnete Domäne D u.U. Zugriff auf alle Dateien des Aufrufers und kann darauf unter dem Deckmantel des Dienstprogramms zugreifen. Ein Beispiel für das Falltür-Problem ist, daß Dienstprogramme auf spezielle Parameter (Eingaben) in für den Aufrufer unerwarteter Weise reagieren, z.B. durch Benachteiligung bestimmter Benutzer bei der Prozessorvergabe. In beiden Fällen kann die Verwendung kleiner Schutzdomänen das Mißbrauchsrisiko mindern, wenn der Umfang der Domänen sich nach der jeweiligen Aufgabe richtet. Damit wird deutlich, daß nur durch das Zusammenspiel von Strukturierungsmaßnahmen und Zugriffskontrollen die genannten Probleme lösbar sind.

Die Sammlung von Daten in elektronischen Rechenanlagen ermöglicht es einzelnen Benutzern, große Datenmengen in relativ kurzer Zeit zu überfliegen ("browsing") und zu kopieren ("copying"). Dies ist nur möglich, wenn dem Benutzer sehr viele Rechte auf große Objektmengen zur Verfügung stehen. Auch hier kann mit der Einschränkung der Schutzdomänen die Ausnutzung des leichten und schnellen Objektzugriffs kontrolliert werden.

Durch die im Modell enthaltenen Schutzoperationen sind auch Änderungen der Schutzdomänen beschreibbar. Diese Änderungen sind selbst wieder zu kontrollieren und dazu als Schutzprobleme aufzufassen. Das Problem des Rückrufs ("revocation") beschreibt die Entfernung von zuvor weitergegebenen Rechten aus Schutzdomänen. Mit dem Verteilungsproblem ("distribution") ist die Kontrolle über die Ausbreitung von Rechten in einer Schutz-

matrix gemeint und ist damit identisch mit dem als im Modell nicht entscheidbar erkannten Sicherheitsproblem. Schließlich wird als spezielles Verteilungsproblem die Anhäufung ("aggregation") von Rechten in einer Domäne unterschieden. Die Größe einer Schutzdomäne gibt Hinweise auf mögliche Gefahren durch Überfliegen, Kopieren und Anhäufung.

Take-Grant-Modell

Im Take-Grant-Modell (/SNYD77/) werden Graphen zur Darstellung von Zugriffsschutzproblemen verwendet. Ein Schutzsystem wird durch einen endlichen, gerichteten, kantenmarkierten Graphen repräsentiert.

$$GS2 = \{Kno, Kan, M\}$$

$Kno = SUOUSO$; S, O, SO paarweise disjunkt;

$$Kan \subseteq Kno \times Kno \times M$$

$$M = \{\{t\}, \{g\}, \{tg\}\}; t: "take", g: "grant"$$

Es werden drei Knotenarten unterschieden:

- Objektknoten O (Kreis)
- Subjekt- oder Objektknoten SO (gekreuzter Kreis)
- Subjektknoten S (gefüllter Kreis)

Die Kanten können mit $\{t\}$, $\{g\}$ oder $\{t,g\}$ beschriftet sein. Eine Markierung einer Kante (a, b, α) bedeutet, daß a das Recht α für b hat. Die Kantenrichtung ist durch die Reihenfolge der Komponenten (von der ersten zur zweiten Komponente) gegeben. Ein so gegebener Graph heißt Schutzgraph (Bild 8).

Analog zu den Operationen SOP des Modells $GS1$, die Änderungen der Schutzmatrix definieren, werden Änderungen eines Schutzgraphen durch folgende vier Graphersetzungsgesetze definiert. (Seien x, y, z Knoten, paarweise verschieden; α, γ seien Markierungen; " $=>$ " bedeute den Übergang zwischen zwei Graphen durch Anwendung einer Ersetzungsregel.)

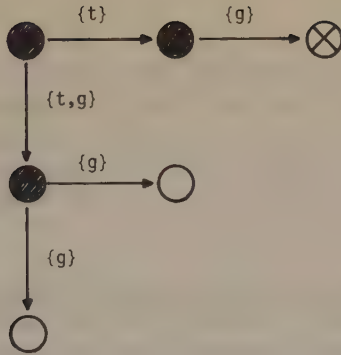
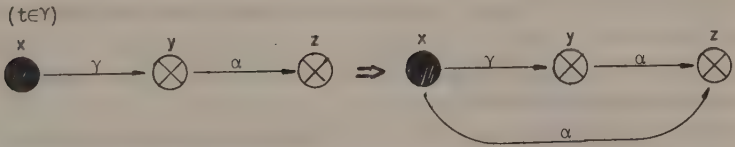
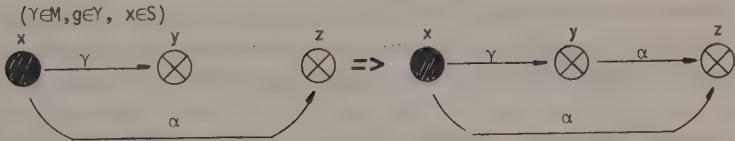


Abb. 8: Beispiel eines Schutzgraphen

- Take-Regel:



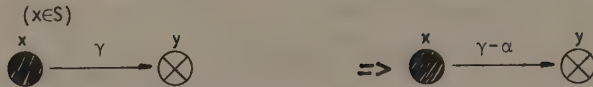
- Grant-Regel:



- Create-Regel:



- Remove-Regel:



Die Endlichkeit des Graphen bedeutet gleichzeitig die Endlichkeit der Mengen S , O , SO , d.h. der Subjekte und Objekte. Während in GS1 die in einem Element der Schutzmatrix enthaltenen Rechte über ihre Weitergabe entscheiden, ist die Veränderung des Schutzgraphen abhängig von der Knotenart (nur Subjekte dürfen Änderungen vornehmen) und abhängig von der Schutzumgebung des Subjekts (die Schutzumgebung eines Knotens ist die Menge der vom Knoten ausgehenden Kanten und Kantenzüge).

Zwei wichtige Ergebnisse wurden bei der Untersuchung von Schutzgraphen gewonnen:

- Das folgende Problem ist in linearer Zeit entscheidbar:

"Ist ausgehend von einem Graphen G durch endlich viele Anwendungen der genannten Ersetzungsregeln ein Graph G' erreichbar, der die Kante (x, y, α) enthält, wenn $x \in S$, $y \in \text{Kno} - \{x\}$ und $\alpha \in CM$?"

- Unter geeignet zu treffenden Voraussetzungen, die nicht genauer ausgeführt werden sollen, kann zwischen zwei Subjekten p und q immer eine Kante mit Markierung $\alpha \in CM$ durch endlich viele Anwendungen der vier Ersetzungsregeln erzeugt werden.

Obwohl also das Sicherheitsproblem, wie oben formuliert, in GS2 in linearer Zeit entscheidbar ist, ist das Modell nicht fein genug, um allgemeine Schutzprobleme zu modellieren, da die Rechteverteilung zwischen den Subjekten des Schutzsystems nur mit starken Einschränkungen (Trennung der Subjekte) kontrollierbar ist. Deshalb wurden z.B. in /JONE78a/ Vorschläge zur Erweiterung des Modells gemacht, die aber nur bez. ihrer Anwendbarkeit auf Schutzprobleme, nicht bez. ihrer formalen Eigenschaften untersucht wurden.

In /SNYD81/ wird gezeigt, wie die Rechtemenge $\{t, g\}$ formal um Rechte für Operationen ("read", "write", "append") erweitert werden kann. Dabei sind die Ergebnisse des einfachen Take-Grant-Modells übertragbar.

1.1.4 Das Informationsflußmodell von Denning

Schutzsysteme, die den Fluß von Informationen zwischen Objekten kontrollieren, werden nicht mit den vorgestellten Zugriffsschutzmodellen dargestellt, da sich der Schutzzustand nicht durch explizite Schutzoperationen (SOP, Graphersetzungsgesetze), sondern aus dem Schutzzustand der beteiligten Objekte bestimmt. Während Zugriffsschutzmodelle beliebig strukturierte ("discretionary") Schutzprobleme beschreiben, dienen Informationsflußmodelle zur Beschreibung fest strukturierter ("mandatory") Schutzprobleme. Ursprünglich entstanden diese Informationsflußprobleme aus der Entwicklung von Rechenanlagen für militärische Anwendungen. Sie sind charakterisiert durch das militärische Sicherheitsmodell, das aus mehreren, linear geordneten Klassifikationen besteht. Informationsflüsse dürfen nur von niedriger zu höher klassifizierten Objekten stattfinden. Häufig wird das Modell um eine Abteilungsmenge ("compartment") erweitert mit der Einschränkung, daß Informationen nur von Mengen zu ihren Obermengen fließen dürfen. Informationsfluß selbst wird von den ausführbaren Operationen bestimmt. Vorteil der Informationsflußmodelle ist ihre Anwendung in der Verifikation, da durch Zuordnung von Informationsflüssen zu den Anweisungen einer Programmiersprache statische Untersuchungen durchgeführt werden können.

Das Verbandsmodell von Denning (/DENN76/) ist ein 5-Tupel

$$GS3 = (S, O, SC, \oplus, \rightarrow)$$

S ist die Menge der Subjekte, $O \cup S$ die Menge der Objekte. Objekte können z.B. die Variablen eines Programms sein. SC ist die Menge der Sicherheitsklassen. $\oplus$ bezeichnet den lub-Operator ("least upper bound"), der für je zwei Operandenklassen die resultierende Klasse bei ihrer Verknüpfung angibt ("--->" be-

zeichne im weiteren den Abbildungspfeil bzw. die logische Implikation, im Gegensatz zur Informationsflußrelation " $\rightarrow$ ".):

$\oplus : SC \times SC \rightarrow SC$

" $\rightarrow$ " heißt Flußrelation und definiert die möglichen Informationsflüsse in GS3, die in Verknüpfungen der Operanden vorkommen dürfen. Ist das Ergebnis "false", ist die Verknüpfung nicht ausführbar:

$\rightarrow : SC \times SC \rightarrow \{true, false\}$

Das System GS3 heißt dann sicher, wenn keine Folge von Operationen zu Informationsflüssen führen kann, die der Flußrelation widersprechen.

In ihrer ursprünglichen Arbeit begründet D.Denning die Annahme, daß $(SC, \oplus, \rightarrow)$ einen Verband bildet, mit Eigenschaften der zu modellierenden Schutzprobleme. Zur Veranschaulichung diene folgendes Beispiel (Bild 9).

$SC = \{ \text{unbeschränkt, vertraulich, geheim, streng geheim} \}$

Seien $i, j, k \in SC$. Dann gilt für $\oplus$: $k = i \oplus j \iff k = \max(i, j)$
und für $\rightarrow$: $i \rightarrow j \iff i \leq j$

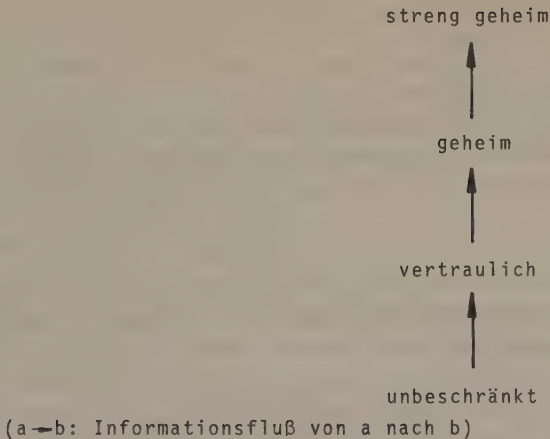


Abb. 9: Militärisches Sicherheitsmodell als Verband

Da aus der Verbandstheorie bekannt ist, daß das Kreuzprodukt zweier Verbände wieder einen Verband ergibt, kann das eingangs erwähnte militärische Sicherheitsmodell um die Abteilungen Heer, Marine und Luftwaffe erweitert werden.

$SC = \{ \text{unbeschränkt, vertraulich, geheim, streng geheim} \} \times \{ \text{Heer, Marine, Luftwaffe} \}$

Seien $(i_1, i_2), (j_1, j_2), (k_1, k_2)$ aus SC .

Dann gilt für $\oplus$: $(k_1, k_2) = (i_1, i_2) \oplus (j_1, j_2) \iff k_1 = \max(i_1, j_1) \wedge k_2 = i_2 \cup j_2$

und für $\rightarrow$: $(i_1, i_2) \rightarrow (j_1, j_2) \iff i_1 \leq j_1 \wedge i_2 \subseteq j_2$

Der in Bild 10 graphisch dargestellte Verband mit der angegebenen Flußrelation dient vor allem im militärischen Bereich zur Modellierung von Informationsflüssen (ADEPT-50, Abschn. 1.2). Aber auch in Firmen ist das Modell anwendbar (Abteilungen, Unternehmensbereiche).

Mit Hilfe des Informationsflußmodells sollen einige weitere Schutzprobleme vorgestellt werden. Das Problem der Anhäufung (1.1.3) wird durch die Sicherheitsklassen neu definiert. Bei dynamischen Änderungen der Klassen kann sich durch Anwendung des $\oplus$ -Operators die Sicherheitsklasse eines Subjekts erhöhen, so daß anfänglich verbotene Informationsflüsse zum Subjekt mög-

Abkürzungen:

u: unbeschränkt, v: vertraulich, g: geheim, s: streng geheim;
H: Heer, M: Marine, L: Luftwaffe.

$$(i_1, j_1) \rightarrow (i_2, j_2) \iff i_1 = i_2 \wedge j_1 \subseteq j_2$$

$$(i_1, j_1) \dashrightarrow (i_2, j_2) \iff i_1 < i_2 \wedge j_1 = j_2$$

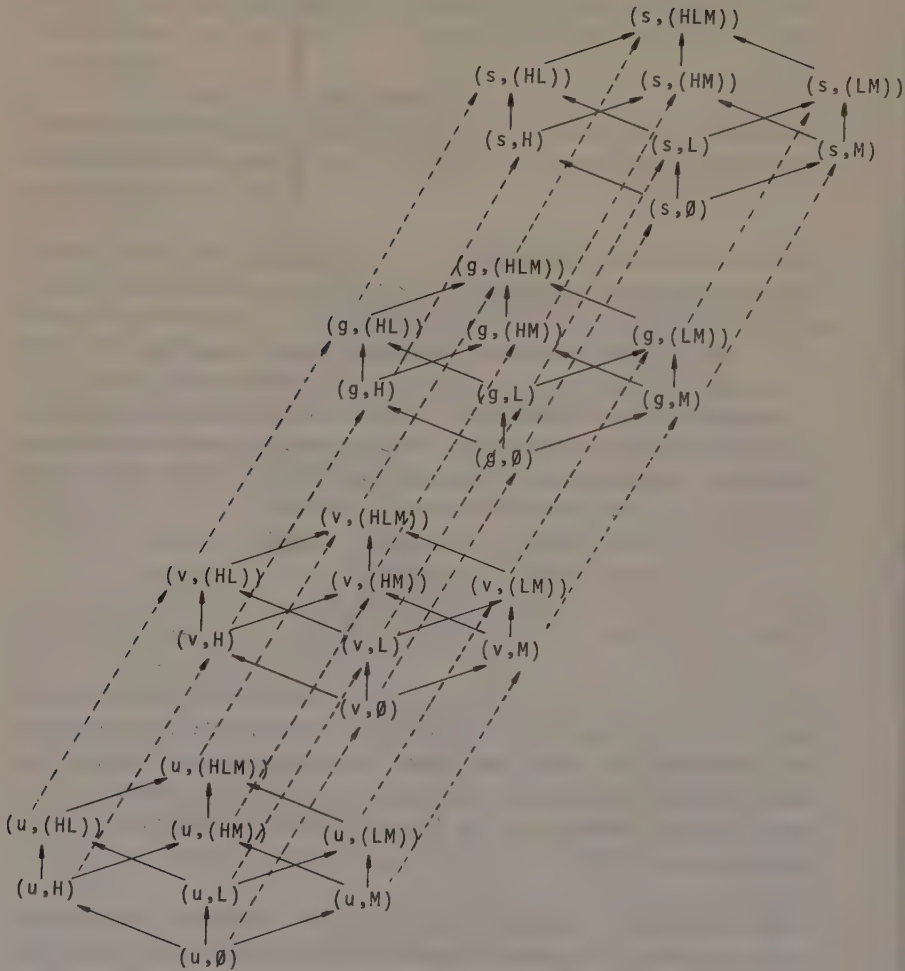


Abb. 10: Erweitertes militärisches Sicherheitsmodell

lich werden. Es muß also eine maximale Sicherheitsklasse für die jeweiligen Subjekte eingeführt werden oder Operationen, die zur Erhöhung der Klasse führen, müssen eingeschränkt werden. Mit der Anhäufung ist das Problem der Rückstufung ("sanitization") verbunden. Dazu muß untersucht werden, wann ein hochklassifiziertes Objekt eine niedrigere Klassifikationsstufe erhalten kann, so daß es für nieder klassifizierte Subjekte wieder lesbar ist.

Ziel von statischen Informationsflußuntersuchungen ist die Analyse von Kanälen zur Informationsübertragung. Um Informationen sichtbar zu machen, können nicht nur bekannte Kanäle wie Programmvariablen, Dateien, E/A-Geräte verwendet werden, sondern auch sogenannte versteckte Kanäle. Darunter sind im Rechen-system vorhandene und sichtbare Objekte und deren Eigenschaften zu verstehen, die nicht in direktem Zusammenhang zum betrachteten Schutzsystem stehen (die formale Definition eines versteckten Kanals erfolgt in Kapitel 3). Versteckt heißen Kanäle, wenn sie nicht als Objekte im Schutzmodell enthalten sind oder damit verbundene Informationsflüsse nicht aus der angegebenen Flußrelation " $\rightarrow$ " ersichtlich sind, d.h. nicht in der transitiven Hülle verborgen sind. In Abhängigkeit vom zur Übertragung verwendeten Medium spricht man von Zeitkanälen (Laufzeit, Wartezeit von Programmen), von Speicherkanälen (Belegung, Freigabe von Speicherbereichen) oder von E/A-Kanälen (Plattenbelegung, Zugriffsverhalten). Das Modell von Denning unterstützt die Entdeckung versteckter Kanäle nur, wenn es sich um Informationsflüsse der transitiven Hülle der Flußrelation " $\rightarrow$ " handelt. Andere Kanäle sind nur durch Analyse des Gesamtsystems erkennbar und damit in das Informationsflußmodell integrierbar. Ein Beispiel ist die Informationsübertragung über die Zwischenräume einer Schnelldruckerausgabe, da diese i.allg. nicht als Objekte zur Informationsübertragung betrachtet werden.

1.1.5 Weitere Modelle

In /JONE78/ wird ein Modell zur Informationsflußanalyse vorgestellt, das einen Zusammenhang zwischen Schutzstrategien und Schutzmechanismen herstellt. Strategie und Mechanismus werden als Funktionen über den Datenbereichen eines Programms definiert:

Mechanismus: $M: D_1 \times \dots \times D_k \rightarrow E \cup F$

Strategie: $S: D_1 \times \dots \times D_k \rightarrow N$

($D_1, \dots, D_k$ Datenbereiche; E, F, N Mengen)

Ein Mechanismus M heißt zuverlässig bez. einer Strategie S , wenn es eine Abbildung M' von N auf $E \cup F$ gibt mit:

$$M(D_1, \dots, D_k) = M'(S(D_1, \dots, D_k))$$

Der Mechanismus M muß sich also so verhalten, wie der Mechanismus M' , der auf dem Wertebereich der Strategie S definiert ist.

Zwischen verschiedenen Mechanismen, die zuverlässig bez. einer Strategie sind, wird eine partielle Ordnung definiert. Die Ordnung ergibt sich aus dem Differenzierungsgrad eines Mechanismus bez. der zulässigen und unzulässigen Informationsflüsse. Damit kann gezeigt werden, daß es zu jeder Strategie einen maximalen Mechanismus gibt. Dieser ist allerdings nicht immer konstruierbar.

Das in /COHE78/ vorgestellte Informationsflußmodell ähnelt der Definition von Informationsübertragung in Kapitel 2. Informationsfluß zwischen zwei Variablen a und b in einem Programm P wird über starke Abhängigkeit ("strong dependency") eingeführt. Dabei wird der Gedanke formalisiert, daß Information zwischen zwei Variablen a und b im Programm P dann übertragen werden kann, wenn Änderungen von a zu Änderungen von b führen können.

1.2 Implementierte Lösungsansätze für Schutzprobleme

Als erster wichtiger Schritt zur Implementation von Schutzmaßnahmen ist ALGOL60 anzusehen, da dort mit dem sprachlichen Hilfsmittel des Blockkonzepts die Einschränkung des Gültigkeitsbereichs und Existenzbereichs von Variablen eingeführt wurde. Es sind statische und dynamische Überprüfungen von Feldgrenzen und der Variablentypen möglich, so daß absichtliche oder unabsichtliche Verletzungen von Feldgrenzen und Typinkompatibilitäten entdeckt werden können. Das Klassenkonzept in SIMULA67 ist eine Erweiterung des Blockkonzepts und erlaubt das Verbergen von Programmteilen. Diese Eigenschaft wurde erst mit der Modularisierungsdiskussion und dem Konzept der abstrakten Datentypen Mitte der siebziger Jahre wieder aktuell. Zu dieser Zeit entstanden Sprachen wie CLU, EUCLID, MODULA, die meist zur Implementierung von Betriebssystemen verwendet werden und deshalb den dortigen Verhältnissen angepaßt sind. Wesentlich ist jeweils ein Modularisierungskonzept, das die statische Trennung und Kapselung zusammengehöriger Programm- und/oder Datenteile erlaubt. Zu der Trennung verschiedener Programmteile kommt die Aufteilung in Spezifikations- und Implementationsteile. Mit dem Spezifikationsteil wird der sichtbare Teil eines Moduls beschrieben, während der Implementationsteil für externe Moduln verborgen bleibt. Zum Import in anderen Moduln genügt die übersetzte Schnittstellenbeschreibung. Diese Ideen werden speziell in Ada und Modula-2 berücksichtigt.

Die Behandlung von Schutzproblemen in Betriebssystemen geht auf die Entwicklung des Mehr-Benutzer-Mehr-Programm-Betriebs zurück. Wegen der konkurrierenden Benutzung von Ressourcen einer Rechananlage und dadurch möglicher gegenseitiger Störungen der Programme müssen verschiedene Benutzer voneinander getrennt werden. Obwohl Zugriffsschutzprobleme gemeinsam mit Synchronisationsproblemen entstanden, wurden zunächst nur Synchronisationsprobleme systematisch untersucht, weil ohne Synchronisationsmaßnahmen auch bei kooperativen Benutzern die Konsistenz mehrfach genutzter Daten nicht gewährleistet werden kann. Besondere Anstrengungen zur Integration von Zugriffsschutzmaßnahmen in ein Betriebssystem wurden zuerst beim System

ADEPT-50, das für militärische Anwendungen konzipiert war, gemacht. Dabei wurden den im Betriebssystem verwalteten Objekten (Programme, Daten, Geräte) Klassifikationen zugeordnet und der Informationsfluß zwischen den Objekten kontrolliert. Eine ähnliche Vorgehensweise findet sich auch in MULTICS. Obwohl dann noch verschiedene experimentelle Betriebssysteme entwickelt wurden (HYDRA, PSOS), war ihr Einfluß auf kommerziell verfügbare Betriebssysteme gering. Lediglich Paßwortmechanismen (Rechtemechanismen, die den Zugriff auf fest vorgegebene Objektklassen kontrollieren) wurden allgemein akzeptiert und in die Systeme integriert. Die fehlenden Mechanismen sollten in den Systemen "Secure UNIX" (/POPE79/), KSOS (/BERS79/, "kernelized secure operating system") durch statische Informationsflußanalysen und durch Struktureigenschaften ausgeglichen werden.

Auch im Bereich der Hardware war im System ADEPT-50 erstmals ein einfacher Schutzmechanismus integriert. Das System VM/370 von IBM enthielt durch die virtuelle Adressierung Möglichkeiten zur Überprüfung von Segmentzugriffen. PLESSEY-250 gilt als erstes kommerziell verfügbares System, das einen Capabilitymechanismus enthält. Damit ist die Verbindung von Objektnamen und ausführbaren Operationen (=Rechten) gemeint. Der Cambridge CAP-Computer, der ebenfalls die Adressierung und Rechtekontrolle über Capabilities unterstützte, wurde nur experimentell genutzt. Erst in letzter Zeit ist mit dem System /38 von IBM eine Hardware-Entwicklung, die die Capability-Adressierung unterstützt, auf dem Markt. Gängige Schutzmaßnahmen wie der privilegierte Befehlsmodus oder die Kontrolle von Speicherzugriffen sind aber auch in Mikrocomputern schon verfügbar (iAPX 286, Z8000). Abb. 11 zeigt eine zeitliche Einordnung der verschiedenen Entwicklungen.

Bei der Implementation von Schutzmaßnahmen genügt es nicht, nur einen der drei Bereiche Programmiersprache, Betriebssystem oder Hardwarearchitektur zu betrachten. Bild 12 veranschaulicht die dabei entstehenden Probleme. Bei der Reduktion der Schutzmaßnahmen auf Hardwarekomponenten können nur allgemein anwendbare Mechanismen zur Verfügung gestellt werden, deren Verwen-

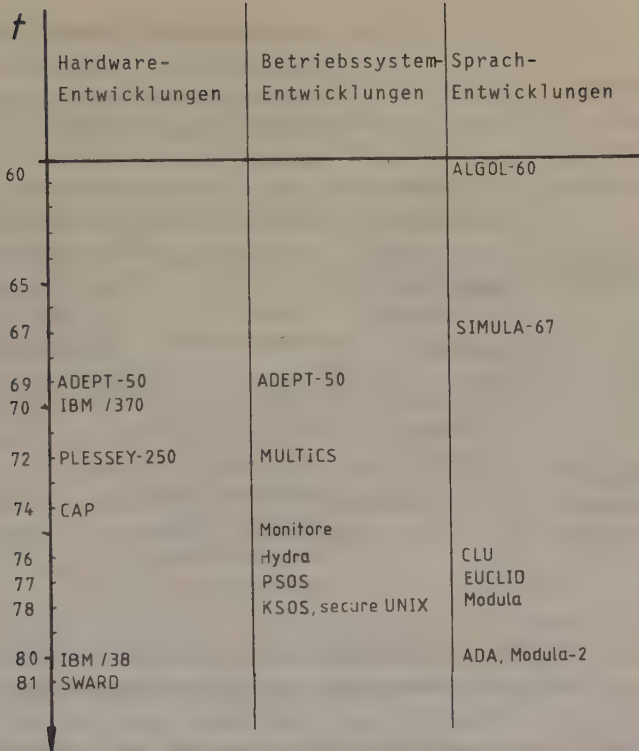


Abb. 11: Zeitliche Einordnung bekannter Lösungsansätze für Schutzmaßnahmen

dung durch die darauf aufbauende Software erzwungen werden muß. Analog verhält es sich bei Reduktion von Schutzmaßnahmen auf das Betriebssystem. Zusätzlich muß durch eine geeignete Programmiersprache der direkte Zugriff auf die zugrunde liegende Maschine ausgeschlossen werden. Die rein programmiersprachliche Lösung von Schutzproblemen verbietet sich wegen der dabei nicht berücksichtigten dynamischen Fehlermöglichkeiten. Weitere Bereiche eines Rechensystems wie Datenbanksysteme oder Kommunikationssysteme werden nicht behandelt, da sie entweder auf den zur Verfügung gestellten Mechanismen und Konstrukten der genannten drei Bereiche basieren (Zugriffskontrolle, Synchronisa-

Programmiersprache		Durch Assemblerprogrammierung kann das Betriebssystem umgangen werden	<i>Schutzkonstrukte der Sprache; Mechanismus im Laufzeitsystem</i>
Betriebssystem	Der Mechanismus muß nicht verwendet werden und ist damit umgebar	<i>Schutzmechanismus im Betriebssystem</i>	Durch Ausnutzung von Betriebssystemeigenschaften kann das Laufzeitsystem umgangen werden
Hardware-Architektur	<i>Schutzmechanismus "verdrahtet"</i>	Fehlende Hardwareunterstützung für den Mechanismus; langsam, keine Berücksichtigung von Software-Fehlern	

Abb. 12: Zuordnung von Schutzmaßnahmen zu den Bereichen Hardware, Betriebssysteme und Programmiersprachen

tion) oder davon unabhängige Maßnahmen für den jeweiligen Problembereich verwenden (Inferenz, Leitungsspionage).

Nachdem im ersten Abschnitt grundlegende Begriffe eingeführt wurden, sollen nun konkrete Lösungsansätze für Zugriffsschutzprobleme durch spezielle Architekturen, Betriebssystementwicklungen und Sprachkonstrukte betrachtet werden. Zur Veranschaulichung werden die Implementationen mit Hilfe der eingeführten Modelle definiert. Obwohl dazu auch Informationsflußmodelle verwendet werden, wird von Zugriffsschutzmechanismen gesprochen, da nicht die Information selbst, sondern der Zugriff auf die entsprechenden Behälter kontrolliert wird. Eine ausführliche Darstellung gibt /MERB80/. Der dritte Abschnitt enthält bekannte Möglichkeiten zur Spezifikation von Schutzproblemen.

1.2.1 Zugriffsschutzmechanismen in Rechnerarchitekturen

Segment- und Seitentabellen bzw. die zugeordneten Register können als die ersten Schutzmechanismen in Rechnerarchitekturen angesehen werden, obwohl sie in erster Linie die Speicherverwaltung und Adressierung in Mehr-Programm-Systemen unterstützen sollen. Damit sind die Adressierungsbereiche verschiedener Programme gegeneinander abgrenzbar. Durch die Verwendung von Grenzregistern können auch Adreßbereichsüberschreitungen festgestellt werden. Für allgemeine Schutzsysteme ist aber der Feinheitgrad der unterscheidbaren Speicherobjekte zu gering, da die zu schützenden Einheiten (einzelne Seiten oder Segmente) größer als 1 K-Byte sind. Neben der Adreßrechnung sind auch Zugriffskontrollen durchführbar, indem ein der Speichereinheit zugeordnetes Recht mit dem jeweiligen Operationsstatus verglichen wird. So enthält z.B. das System/360 ein Schreibrecht, womit jeder schreibende Speicherzugriff auf Zulässigkeit überprüft werden kann.

Das System ADEPT-50, eine Erweiterung des Systems/360-50, enthält neben dem Schreibrecht auch ein Leserecht ("fetch"), das die Zulässigkeit lesender Speicherzugriffe kontrolliert. Darüber hinaus kann durch eine vier Bit breite Hardware-Maske eine von 15 verschiedenen Benutzerklassen einzelnen Speicherseiten zugeordnet werden. Eine Maske mit vier Nullbits erlaubt den Zugriff auf den gesamten Speicher, während sonst die Maske mit der der Seite zugeordneten Benutzerklasse übereinstimmen muß. Um den Zugriff auf die Schutzinformation zu kontrollieren, unterscheidet die Maschine zwei Zustände: Überwacher- und Problemzustand ("supervisor-, problem-state"). Im Problemzustand ist nur eine Teilmenge aller Befehle ausführbar. Insbesondere sind solche Befehle dem Überwacherzustand vorbehalten, die den Zustand der Hardware-Maske ändern. Der Wechsel zwischen Überwacher- und Problemzustand ist nur unter besonderen Bedingungen (Unterbrechungen, SVC) möglich und wird von der Hardware kontrolliert. /MOLH70/ konnte zeigen, daß die Hardware-Maßnahmen trotz der Kontrollen umgangen werden können.

Der realisierte Schutzmechanismus ist in Form einer Schutz-

matrix SM mit einer Zeile darstellbar, wobei die Objekte Seiten des virtuellen Adreßraums sind und die Subjekte einer der 16 Benutzerklassen angehören. Die Rechte "fetch", "write" können und eine der Benutzerklassen muß in einem Matrixelement enthalten sein.

Sei $(i, j, (r, k)) \in SM$ mit $r \in \{\text{fetch}, \text{write}\} \wedge k \in \{1, \dots, 15\}$

Dann hat ein laufender Prozeß mit Klasse i , $0 \leq i \leq 15$ das Lese-recht (Schreibrecht) für Objekt j , falls

$$(i=0 \vee i=k) \wedge \text{fetch} \in r \ (\text{write} \in r)$$

Bild 13 zeigt einen möglichen Zustand der Schutzmatrix im System ADEPT-50.

Klasse des laufen- den Prozesses	Seitennummern					
	0	1	2	3	4	...
2	1 fw	15 w	0 fw	0 f	2 w	

Abb. 13: Schutzmatrix für das System ADEPT-50

Das Konzept der Maschinenzustände wird im System MULTICS auf acht Schutzringe erweitert, die Zustandswechsel werden durch spezielle Hardware überprüft. Zusätzlich werden den Objekten (Segmenten) Rechte zugewiesen, die das Lesen, Schreiben oder Ausführen gestatten. Die Schutzinformation wird in Deskriptoren (Bild 14) abgelegt, die den Segmentidentifikator, die zulässigen Rechte, Klammern und Tore enthalten. Klammern und Tore steuern den Übergang zwischen Schutzringen (s.u.). Jedem Prozeß (Subjekt) ist eine Menge von Deskriptoren, seine Schutzumgebung, zugeordnet. Die Deskriptoren können nicht an den Objekten stehen, da sonst für alle Subjekte die gleichen Zugriffseinschränkungen gelten, wie es in ADEPT-50 der Fall ist.

Die Überprüfungen beim Zugriff auf ein Segment hängen von der auszuführenden Operation ab:

- 1) Lesen: Die aktuelle Ringnummer ist kleiner als Klammer L2 und die Rechte enthalten das Leserecht.

Segment- nummer	Länge	r_1	r_2	r_3	l_1	l_2	l_3	Tore
--------------------	-------	-------	-------	-------	-------	-------	-------	------

r_1, r_2, r_3 : Rechte

l_1, l_2, l_3 : Klammern

Abb. 14: Deskriptor in MULTICS

- 2) Schreiben: Die aktuelle Ringnummer ist kleiner als Klammer L_1 und die Rechte enthalten das Schreibrecht.
- 3) Ausführen: Die Rechte enthalten das Ausführrecht und
 1. Fall aktuelle Ringnummer kleiner als Klammer L_1
 ---> L_1 wird aktueller Ring, Prozeduraufruf
 2. Fall $L_1 \leq$ aktuelle Ringnummer $\leq L_2$
 ---> Prozeduraufruf
 3. Fall $L_2 <$ aktuelle Ringnummer $\leq L_3$
 ---> Prozeduraufruf über ein Tor, L_2 wird aktueller Ring
 4. Fall $L_2 <$ aktueller Ring
 ---> Zurückweisung

Die Klammern ändern damit die Schutzumgebungen der Subjekte bei Prozeduraufrufen. Die Definition von Toren garantiert, daß bei Erweiterung von Schutzumgebungen die Aufrufe stets über definierte Einsprungstellen erfolgen, so daß dort weitere Überprüfungen, z.B. der Parameter, möglich sind. Die Hardware-Realisierung soll den Umgebungswechsel beschleunigen, so daß keine Verzögerungen gegenüber anderen Prozeduraufrufen eintreten.

Die genannten Mechanismen (Rechteüberprüfung, Zustandswechsel) sind inzwischen in Mikroprozessorsystemen (Z8000, iAPX 286) verfügbar. So bietet die Zentraleinheit iAPX 286 die Unterscheidung vier verschiedener Zustände, der Übergang zwischen den Zuständen wird automatisch überwacht. Der Zugriff auf Speichersegmente wird auf Zulässigkeit überprüft. Dabei wird zwischen drei Rechten ("lesen", "schreiben", "ausführen") unterschieden. Zusätzlich wird bei jedem Segmentzugriff die mögliche Überschreitung der Segmentgrenze untersucht. Die dazu notwendigen Segmentinformationen werden in Registern zur schnellen Ab-

arbeitung hinterlegt und sind nur in einem speziellen Zustand (privilegierter Zustand) änderbar. Sie sind jeweils einem Prozeß zugeordnet und werden bei Prozeßwechseln automatisch ausgetauscht. Durch die Ausführung der Prüfvorgänge parallel zur Adressierung in der Zentraleinheit wird die Zykluszeit nicht verlängert.

Rechteüberprüfung und Zustandswechsel können als grundlegende Elemente eines Schutzmechanismus verstanden werden, da sie bestehende Schutzeinschränkungen auf Speicherzugriffe kontrollieren. Damit ist eine effiziente dynamische Kontrolle möglich. Sie reichen aber nicht aus, um allgemeine Schutzkonzepte effizient zu implementieren, da z.B. der Mechanismus des Systems ADEPT-50 die möglichen Subjektklassen beschränkt und Zugriffsunterschiede mehrerer Subjekte auf ein Objekt nur durch die Benutzerklasse ausdrückbar sind. Mögliche Erweiterungen des Mechanismus durch Software-Komponenten bleiben dabei unberücksichtigt, da zunächst von einem möglichst allgemeinen, in Hardware realisierten Mechanismus gesprochen wird.

Capability-Mechanismen

Fabry (/FABR74/) schlägt als allgemeinen Mechanismus sowohl zur Adressierung, insbesondere von gemeinsamen Objekten mehrerer Prozesse, als auch zur Zugriffskontrolle den Capability-Mechanismus vor, der zum ersten Mal im System PLESSEY-250 (/ENGL72/) verwirklicht wurde. Die Idee geht ursprünglich auf /DENN66/ zurück. Eine Capability ist ein eindeutiger Name für ein Objekt (nicht notwendig ein Speichersegment) mit dem Rechte verbunden sind. Die Rechte geben im Zusammenhang mit dem durch den Namen bezeichneten Objekt Auskunft über die ausführbaren Operationen des Subjekts, das die Capability besitzt. Die Realisierung der "besitzt"-Relation ist abhängig von der Realisierung des Mechanismus. Capabilities werden durch folgende Eigenschaften charakterisiert (/NEED80/):

- 1) Capabilities sind nicht beliebig änderbar
- 2) Capabilities sind gültig durch ihre Existenz
- 3) Capabilities sind übertragbar

4) Capabilities sind selbstidentifizierend

Eine fünfte Eigenschaft, die Eindeutigkeit einer Capability, wird wegen der besonderen Probleme, die die Erfüllung der Bedingung mit sich bringt, nicht gefordert. Eigenschaft 1 verlangt, daß die Erzeugung und Änderung von Capabilities nur speziellen Programmen vorbehalten bleibt, da sonst die weiteren Eigenschaften nicht erfüllbar sind. Eine Realisierungsmöglichkeit ist die Beschränkung der Änderungen auf privilegierte Zustände, die nur für besondere Systemroutinen erreichbar sind. Eigenschaft 2 unterstützt die effiziente Verwendung von Capabilities, da zum Objektzugriff nur die vorgelegte Capability untersucht werden muß. Mit Eigenschaft 3 werden gemeinsame Objekte zwischen mehreren Subjekten einfach realisierbar. Schließlich wird mit Eigenschaft 4 ebenfalls die Adressierung von Objekten vereinfacht.

Gegeben sei die Schutzmatrix $SM \in S \times O \times R_{\text{pot}}$, $R = \{r_1, \dots, r_n\}$
 Außerdem folgende Capabilitymengen eines Subjekts i:

$$C_i = \{(o_1, R_1), \dots, (o_m, R_m)\}, o_j \in O \wedge R_j \subset R$$

Dann gilt:

$$(\forall (s, o, R') \in SM) (\exists (o', R'') \in C_i) (o' = o \wedge R'' = R')$$

Für jedes Element (i, j, R') einer Matrixzeile, die dem Subjekt i zugeordnet ist, muß es eine Capability aus der Capability-Menge C_i geben, die auf das der Spalte j der Matrix SM zugeordnete Objekt zeigt und deren Rechte mit der Menge R' übereinstimmen. U.U. sind in der Capability-Menge mehrere Capabilities für dasselbe Objekt enthalten.

Obwohl in mehreren experimentellen Systemen die Vorteile des Capability-Konzepts gezeigt werden konnten (HYDRA, PSOS), wurde es zunächst nur im CAP-System durch Hardware-Elemente und Mikroprogramme unterstützt. Erst Anfang der 80er Jahre ist mit dem System/38 von IBM eine Maschine auf dem Markt, die mit Capabilities arbeitet.

Zur Realisierung von Capabilities gibt es i.allg. zwei Vorgehensweisen. Ziel ist jeweils, die Schutzinformation (Capabilities) von anderen Daten zu trennen. Zum einen können spezielle Speicherbereiche für Capabilities reserviert werden (Capabi-

lity-Listen). Die "besitzt"-Relation eines Prozesses besteht dann aus allen zugänglichen Capability-Listen. Die zweite Methode ist die Verwendung von "tag"-Bits, das ist eine Kennung für die im Speicher enthaltenen Objekte und ihre Typen. Die Anzahl der Bits richtet sich nach den von der Hardware zu unterscheidenden Typen (TR440: zwei Bits, vier Typen). Die "besitzt"-Relation in diesem Fall ergibt sich aus den adressierbaren Capabilities eines Prozesses. Im Cambridge CAP-Computer werden Capability-Listen verwendet (/NEED77/), um keine Speichererweiterungen vornehmen zu müssen. Eine Adresse zum Zugriff auf ein Speicherelement hat die in Bild 15 dargestellte Form.

Zum Speicherzugriff eines Prozesses P sind folgende Schritte

CSnr	Cnr	0
------	-----	---

CSnr: Nummer des Capabilitysegments

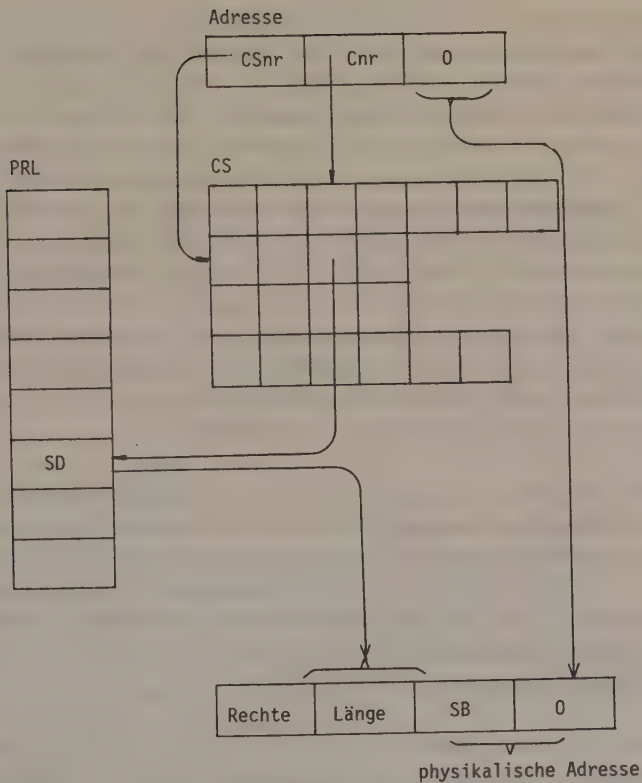
Cnr: Nummer der Capability im Capabilitysegment CS

0: relative Adresse im angesprochenen Segment

Abb. 15: Adresse im CAP-System

notwendig (Bild 16): Zunächst muß im Segment CS die an Stelle C stehende Capability geholt werden. Diese zeigt auf einen Eintrag in der Objekttabelle des Prozesses P ("process resource list", PRL). Dort steht die physikalische Adresse des Segments, die für den Prozeß erlaubten Rechte und die Segmentgrenze. Der Zugriff wird nur ausgeführt, wenn der Operationsstatus zur Rechtemenge paßt und die relative Adresse 0 kleiner oder gleich der Segmentgrenze ist. Anschließend werden physikalische Adresse und relative Adresse konkateniert und zur physikalischen Adressierung verwendet.

Die zur Bearbeitung von Capabilities notwendigen Operationen (Adreßumsetzung, Listebearbeitung, Capability-Manipulationen) sind mikroprogrammiert. Weitergehende Operationen zur Verwaltung der Listen sind im Betriebssystem enthalten und mit den Basisoperationen implementiert. Der Adressierungsmechanismus wird durch Register unterstützt. Einerseits enthalten fünf Ca-



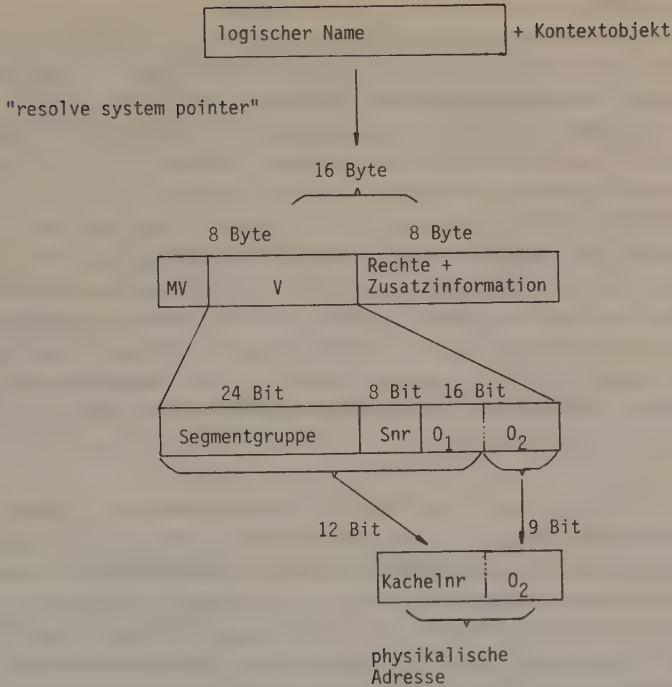
CSnr: Nummer des Capabilitysegments
 Cnr: Nummer der Capability im CS
 0: relative Adresse im angesprochenen Segment
 CS: Capability-Segmente des laufenden Prozesses
 PRL: Objekttafel des laufenden Prozesses
 SD: Segmentdeskriptor
 SB: Segmentbasis

Abb. 16: Adressierung im CAP-Computer

pability-Register die Adressen der durch die Listennummer vorgegebenen Capability-Liste CS. Die Inhalte der Register sind vom aktuellen Prozeß, von der aktuellen Prozedur und dem bearbeiteten Segment abhängig. Zwei Register werden speziell zur Umschaltung bei Prozeduraufrufen verwendet, um Capabilities als Parameter mit eventuell eingeschränkter Rechtemenge weiterzugeben. Um einen Adreßausdruck nicht jedes Mal neu auswerten zu müssen, wird die Hardware um einen Cache-Registersatz mit 64 Registern erweitert. Darin sind die zuletzt berechneten Segmentadressen mit den Zugriffseinschränkungen enthalten. Darüber hinaus wird durch Zusatzinformationen eine Zuordnung zwischen den Cache-Einträgen und den laufenden Prozessen/Prozeduren hergestellt. Dadurch wird die Neuberechnung von absoluten Adressen bei Prozeß- und Prozedurwechseln vermieden.

Die Implementation von Capabilities mit Markierungsbits, die die gemeinsame Abspeicherung von Capabilities und Objekten anderen Typs in denselben Speicherbereichen erlaubt, wird im System/38 (/BERS80/) angewandt. Eine Zielvorstellung des Systems ist, Benutzer, die den Schutzmechanismus verwenden, nicht zu benachteiligen. Zunächst soll wieder ein allgemeiner Adressierungsvorgang beschrieben werden (Bild 17).

Aus dem logischen Namen eines Objekts und einem sogenannten Kontextobjekt wird durch den Mikrobefehl "resolve_system_pointer" ein Zeiger ("pointer") in den globalen, für alle Prozesse gleichen, virtuellen Adreßraum erzeugt. Dieser ist wesentlich größer als der physikalisch realisierte virtuelle Adreßraum, um innerhalb einer bestimmten Lebensdauer des Systems (ca. 10 Jahre) ein mehrfaches Erzeugen des gleichen Zeigers zu vermeiden. Ein Zeiger kann aus maximal drei Teilen bestehen: Der eindeutigen Adresse im virtuellen Adreßraum (64 Bits), den zugehörigen Rechten und der Zusatzinformation (16 Bytes). Zeiger werden von anderen Daten durch ein Markierungsbit unterschieden. Jedes Wort (Wortlänge = 40 Bit), das einen Teil eines Zeigers enthält, hat ein gesetztes Markierungsbit. Das Setzen der Markierung wird von Mikroprogrammen kontrolliert, nicht autori-



MV: mit Software verwalteter Teil der virtuellen Adresse
V: mit Firmware verwalteter Teil der virtuellen Adresse
Snr: Segment innerhalb der Segmentgruppe
O₁O₂ 64 KByte-Segment
O₂: Versatz im angesprochenen Segment, Kacheln zu 512 Byte

Abb. 17: Adressierung im System/38

sierte Änderungen von Zeigern führen zur Löschung der Markierungen. Auch die Kenntnis einer virtuellen Adresse führt nicht zu Schutzverletzungen, da die Generierung von Zeigern (Setzen der Markierung) nicht frei zugänglich ist.

Der Speicher ist in Seiten von 512 Byte eingeteilt. 256 Segmente mit maximal 64 KByte Länge werden in Segmentgruppen zusammengefaßt. Die Hardware stellt eine Adreßbreite von 48 Bit zur Verfügung, so daß 2^{24} Segmentgruppen gleichzeitig adres-

siert werden können.

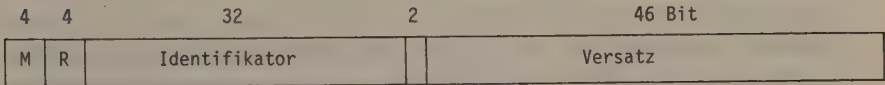
Um eine virtuelle Adreßbreite von 64 Bit zu erreichen, werden 16 weitere Bits in Mikroprogrammen ausgewertet. Dabei wird bei Generierung der physikalischen Adresse überprüft, ob der vom Mikroprogramm bearbeitete Adreßteil der Segmentgruppe zugeordnet ist. Aus 39 Bits der virtuellen Adresse wird durch eine Hashtabelle die zugeordnete Kachel, eventuell nach Laden aus dem Externspeicher, bestimmt und mit dem 9 Bit breiten Versatz innerhalb der angesprochenen Seite konkateniert. Um die Adressierung zu beschleunigen, wird ein Cache-Speicher verwendet, der die zuletzt berechneten Kacheladressen und die zugeordneten virtuellen Adressen enthält. Für laufende Adressierungsvorgänge steht zusätzlich ein Register bereit. Der Cache-Speicher wird ausgehend von der virtuellen Adresse über eine zweite Hash-Funktion angesprochen.

Mit Hilfe eines Mikrobefehls ("create_program") werden aus Benutzerprogrammen (in einer assemblerartigen Zwischensprache) Mikroprogramme erzeugt, die direkt ausführbar sind. Eine Assemblersprache selbst steht nicht zur Verfügung. Dadurch sollen Schutzprobleme, die aus der maschinennahen Programmierung entstehen (Registerzugriffe, Adreßrechnung), vermieden werden. Die Zusicherung, daß Zwischensprachprogramme nur von autorisierten Programmen (Compilern) erzeugt werden, muß auf Software-Ebene erfüllt werden.

Die SWARD-Architektur (/MYER80/) enthält einen ähnlichen Adressierungsmechanismus wie das System/38, erweitert aber die anhand der Markierungen unterscheidbaren Objekttypen auf 16 (vier Markierungsbits). Mit Hilfe der Markierungen können z.B. auch Struktur- und Feldarten gekennzeichnet werden. Jeder Speichereintrag besteht deshalb aus vier Markierungsbits und der nachfolgenden Information, deren Länge vom Typ abhängt.

Eine Capability im SWARD-System besteht aus 88 Bits (Abb.18). Die ersten vier enthalten die Capability-Markierung ("1001"). Anschließend kommen die der Capability zugeordneten Rechte, die jeweils vom adressierten Objekt abhängen. In 32 Bits wird ein eindeutiger Identifikator geführt, der über Hashtabellen auf

das zugehörige Objekt zeigt. Die restlichen 48 Bits dienen zur Adressierung innerhalb eines Objekts (z.B. Strukturkomponenten, Feldelemente), wobei 2 Bits zur Interpretation der folgenden 46 Bits verwendet werden. Die Berechnung von Adressen und die Bearbeitung von Capabilities erfolgt durch Mikroprogramme, analog zum System/38.



M: Typmarkierung

R: Rechte

Abb. 18: Capability der SWARD-Architektur

1.2.2 Schutzmaßnahmen in Betriebssystemen

Die Schutzmaßnahmen in kommerziell verfügbaren Betriebssystemen beschränken sich i.allg. auf die Überprüfung von Speicherzugriffen. Diese werden mit den bereits genannten Mechanismen (Segmentgrenzenüberprüfung; Lese-, Schreib-, Ausführungsrechte; Zustandswechsel) implementiert. Daneben werden spezielle Software-Mechanismen für Dateizugriffe verwendet. Dazu gibt es Paßwortverfahren (z.B. TR440) oder die Definition von Zugänglichkeitsklassen (z.B. "privat;Benutzer1,...,Benutzer n;öffentlich" an der CYBER) und die Unterscheidung zwischen Lese-, Schreib- und ausführenden Zugriffen. Zum Schutz der Benutzer voreinander werden die Benutzerbereiche getrennt (Trennung der virtuellen Adreßräume). Zur Kommunikation zwischen Benutzern stehen entweder spezielle Kanäle zur Verfügung, oder die Kommunikation ist auf gemeinsame Dateien beschränkt.

Auch im kommerziell verfügbaren System UNIX (/RITC74/) basiert der Schutz von Benutzern vor gegenseitigen Zugriffen auf der Trennung der Datenbereiche. Dazu werden die Dateien in Directories baumartig verwaltet, wobei die Baumstruktur von der Benutzermenge abhängt und Dateien verschiedener Benutzer in

verschiedenen Teilbäumen repräsentiert werden. Über die Angabe spezieller Pfadnamen kann aber auch auf Dateien anderer Benutzer zugegriffen werden, da durch den Pfadnamen der Baum in vorgegebbarer Weise durchlaufen werden kann. Die Traversalion eines Dateibaums eines Benutzers kann durch einen Rechtemechanismus eingeschränkt werden, der zwischen dem einer Datei zugeordneten Benutzer (Besitzer), einer ihm zugeordneten Gruppe und den übrigen Benutzern unterscheidet. Der Aufbau der Directory-Bäume schützt den Benutzer vor eigenen fehlerhaften Programmen, wenn die von einem Programm zu bearbeitenden Dateien in eigenen Teilbäumen zusammengefaßt werden. Die Restriktion der Dateizugriffe, aufgeteilt auf 3 Benutzerklassen, schränkt die gegenseitigen Zugriffe von Benutzern grob ein. Die Sicherheit in UNIX ist leicht verletzbar durch die Existenz eines Überbenutzers ("superuser"), der i.allg. für die Wartung des Systems verantwortlich ist. Da dieser nicht neben den Benutzern, sondern bez. der Menge der ausführbaren Operationen über ihnen steht, sind einerseits Fehler des Überbenutzers nicht einzugrenzen. Andererseits kann der irreguläre Erwerb des Status "Überbenutzer" einem nicht autorisierten Benutzer das System frei zugänglich machen.

Das System ADEPT-50

Im Betriebssystem ADEPT-50 (/WEIS69/) wurde zum ersten Mal versucht, ausgehend von einem formalen Schutzmodell, wesentliche Aspekte eines Multiuser-, Multitasking-Betriebssystems zu erfassen und die Schutzüberprüfungen mit einem einheitlichen Mechanismus zu realisieren. Daneben sind weitere, nicht formalisierte Schutzmaßnahmen vorgesehen, die auch einen heute noch üblichen Standard repräsentieren und deshalb beispielhaft beschrieben werden.

Im Modell werden vier Objektarten unterschieden:

- U (Benutzer, "users")
- J (Jobs, "jobs")
- T (Terminals, "terminals")
- F (Dateien, "files").

Jedem Objekt wird ein Profil, bestehend aus

$A = \{a_1, a_2, \dots, a_w\}$ (Berechtigung, "authority")

$C = \{c_1, c_2, \dots, c_p\}$ (Abteilung, "category")

$P \subset U$ (Privileg, "franchise")

zugeordnet. Zwischen den Profilelementen zweier Benutzer (Jobs, Terminals, Dateien) kann eine Ordnungsrelation definiert werden.

Aus vorgegebenen Anfangsbelegungen der Profile und der Abfolge der Änderungsoperationen kann jedes Profil zu einem Zeitpunkt $t > 0$ berechnet werden. Die erlaubten Zugriffe eines Benutzers auf Objekte des Systems werden durch folgende Zusammenhänge bestimmt:

- 1) Ein Benutzer u darf mit dem System arbeiten, wenn er aus der gegebenen Benutzermenge ist.
- 2) Ein Benutzer u darf das Terminal t verwenden, wenn er in den Privilegien des Terminals enthalten ist.
- 3) Ein Benutzer u darf mit Job j auf Datei f zugreifen, wenn die Berechtigung bzw. die Abteilung von j größer oder gleich der Berechtigung bzw. der Abteilung von f ist und wenn u in den Privilegien von f enthalten ist. Beim Zugriff werden die Berechtigung und die Abteilung neu ausgewertet.

Die Implementation macht nicht von allen Profilkomponenten Gebrauch, sondern ist an die Eigenschaften der jeweiligen Objekte angepaßt. Zur Objektidentifikation werden teilweise benutzerdefinierte Namen, teilweise systemeindeutige Bezeichnungen verwendet. Die Dateiidentifikation enthält neben dem Namen noch Informationen über die Dateiart, den erzeugenden Benutzer und das Gerät (Plattenstapel). Es sind vier Berechtigungen definiert: "beliebig", "vertraulich", "geheim", "streng geheim". Die Abteilungsmenge C enthält 16 Abteilungen. Die Privilegien werden nach dem "need-to-Know"-Prinzip, also nach minimalem Bedarf vergeben. Dabei unterstützt das System maximal 500 Benutzer, die bei Initialisierung bekannt sein müssen. Für Terminals werden keine Privilegien unterschieden. Jedem Benutzer werden alle Terminals zugänglich gemacht. Das Privileg eines Jobs enthält den Benutzer, der den Job erzeugt hat.

Die Privilegien einer Datei sind gegenüber denen des Modells erweitert. Neben den Benutzern, die auf die Datei Zugriff haben, wird jeweils ein Attribut über die Zugriffsart geführt. Eine Datei kann privat, halbprivat oder öffentlich sein; Zugriffsattribute sind "lesen", "schreiben", "exklusiv lesen oder schreiben", "lesen oder schreiben".

Eine spezielle Prozedur legt bei Systeminitialisierung die Schutzinformationen für die vorhandenen Objekte (Benutzer, Terminals) an. Die Zugriffskontrolle erfolgt dynamisch, abhängig vom Objekttyp. Eine Prozedur "login" überprüft die Zugangsberechtigung des Benutzers und legt das Schutzprofil des zugehörigen Jobs an. Ein spezieller Systemteil "EXEX" lädt für einen Job solche Programme, die aufgrund des Job-Profiles zugänglich sind. Das geladene Programm läuft unter dem Profil des aufrufenden Jobs, so daß auch Programme mit kleinem Profil sensitive Daten (Daten mit hohem Profil) verarbeiten können. Analog wird bei Dateizugriffen vorgegangen. Der Zugriff wird stets beim Öffnen der Datei durch ein Programm (Profil bestimmt durch den Job) überprüft und auf die beim Öffnen anzugebende Zugriffsart beschränkt. Für die Festlegung des Anfangsprofils einer Datei wird ein heuristisches Verfahren verwendet, das sich aus den Profilverfolgen eines Jobs bestimmt. Damit werden zu niedrige Profileinstufungen zum großen Teil vermieden, während zu große Profile meist am Ende längerer Jobs auftreten.

Die implementierten Schutzmaßnahmen können mit dem Verbandsmodell von Denning beschrieben werden, und zwar als Verband über dem Kreuzprodukt der drei Mengen A, C und P. Der lub-Operator bestimmt sich für A aus dem Maximum zweier Elemente $a \in A$, für C und P aus der Vereinigung zweier Elemente $c \in C$ ($f \in P$). Der mögliche Informationsfluß wird durch die Bedingungen 1 bis 3 definiert.

Die genannten Maßnahmen zur Implementierung des Modells werden durch weitere, nicht im Modell erfaßte Maßnahmen unterstützt. Auf Hardware-Erweiterungen, die z.B. die Aufteilung des Hauptspeichers in maximal 16 Benutzerklassen zulassen, wurde bereits in Abschnitt 1.2.1 hingewiesen. Der Aufruf von Dienst-

programmen des Systems (Testsystem, E/A) wird speziellen Prüfungen unterworfen. Das Löschen von einzelnen Speicherelementen (Hauptspeicher, Platte, Trommel, Band) wird besonders beachtet, um Informationen nicht durch Neubelegung benutzter Speichereinheiten sichtbar zu machen. Durch spezielle Kommandos kann, nach entsprechender Prüfung, die Schutzinformation geändert werden (siehe 1.1, "sanitization"). Schließlich ist es möglich, über die Verwendung einer ausgewählten Operationenmenge Buch zu führen.

Auch im MULTICS-System (/SALT74/) wird, aufbauend auf einem Hardware-Mechanismus zum Schutz von Hauptspeichersegmenten, im Betriebssystem ein Modell zum Schutz von Objekten vorgegebenen Typs realisiert. Mögliche Typen sind Segmente, Nachrichtenwarteschlangen, Directories und Externspeicherdeskriptoren. Jedem Objekt wird eine Zugriffskontrollliste zugeordnet, die die zugriffsberechtigten Benutzer und die ihnen zugeteilten Zugriffsarten bestimmt. Während die Zugriffsarten typabhängig sind, gilt für die Festlegung der zugriffsberechtigten Benutzer eine dreigeteilte Struktur:

Name.Projekt.Abteilung

Das Kreuzprodukt der Namen, Projekte und Abteilungen ist lexikalisch geordnet.

Das grundlegende Schema der Zugriffskontrolllisten wird durch verschiedene weitere Maßnahmen unterstützt. Insbesondere soll die Verwendung von Schutzvorkehrungen für Benutzer nicht unnötig kompliziert oder aufwendig sein. Außerdem ist die Verwendung von Schutzmaßnahmen als Kann-Regel gedacht. Daraus ergibt sich, daß besondere Aufmerksamkeit der Benutzerschnittstelle gewidmet ist. Zunächst werden bei der Erzeugung von Objekten möglichst "natürliche" Vorbelegungswerte verwendet. Ein Beispiel ist, daß für jedes erzeugte Objekt eine initiale Zugriffskontrollliste erzeugt wird, die sich aus der Umgebung bestimmt, in der ein Objekt erzeugt wird. Die Objekte des Systems werden in hierarchisch angeordneten Directories verwaltet. Aus dieser Anordnung ergeben sich für sehr hoch autorisierte Personen große Manipulationsmöglichkeiten. Deshalb werden Zugriffe über die Wurzeln der Directory registriert. Die in 1.2.1 be-

sprochenen Hardware-Erweiterungen (Schutzringe, Klammern) werden zur Trennung von Adreßbereichen der Prozeduraufrufe eines Prozesses verwendet. So können z.B. Dienstprogramme, die anderen Benutzern bereitgestellt werden, auf Teile des benutzereigenen Adreßbereichs eingeschränkt werden oder spezielle Prüfungen vorgenommen werden (Klammern, Tore).

Für den Zugriff auf Systembereiche spielt der Identifizierungsprozeß eine große Rolle, um die Zugriffskontrolllisten den Benutzern korrekt zuzuweisen. Deshalb wird der Identifizierung besondere Sorgfalt gewidmet. Ein übliches Paßwortschema speichert die Paßwörter verschlüsselt ab, so daß ein möglicher Zugriff auf die Liste der Paßwörter nicht mißbraucht werden kann. Eine spezielle Prozedur stellt aussprechbare Zeichenfolgen von acht Zeichen bereit, um benutzereigene Paßwörter, die oft erraten werden können, nicht verwenden zu müssen. Das systematische Erraten von Paßwörtern wird behindert durch die Unterbrechung von Verbindungswegen nach mehreren erfolglosen Versuchen. Außerdem werden Fehlversuche in einem Logbuch vermerkt. Zum Erkennen erfolgreicher Penetrationsversuche wird bei Gesprächsbeginn Datum und Uhrzeit des letzten Gesprächs ausgegeben. Der Eintritt ins System ist nur über Terminals und dem damit verbundenen Identifikationsalgorithmus möglich. An anderen Geräten (z.B. Kartenlesern) eingelesene Programme müssen durch bereits im System aktive Prozesse gestartet werden.

Das System HYDRA

Im Betriebssystem HYDRA (/WULF76/) wird versucht, aufbauend auf einem einheitlichen Schutzmechanismus (Capability-Mechanismus), ein geschütztes System zu entwickeln. Der Mechanismus wird nicht von Hardware unterstützt, so daß das System nur zu experimentellen Zwecken verwendet werden kann. Wesentlich ist die strikte Trennung zwischen Strategie und Mechanismus, die nicht nur in den Schutzmaßnahmen angewandt wird. Wie bereits erläutert, bietet der Capability-Mechanismus eine allgemeine Basis zur Lösung von Zugriffsschutzproblemen. Darauf werden verschiedenartige Strategien implementiert.

Grundlegend in HYDRA ist die Verwendung von Capabilities. Auf der eindeutigen Benennung von Objekten und den darauf zulässigen Funktionen bauen die Operationen eines Betriebssystemkerns auf. Auf den vom Kern bereitgestellten Datentypen und Operationen basieren weitere Betriebssystemteile und Benutzerprogramme. Alle Objekte des Betriebssystems (Dateien, Prozesse, Prozeduren, Segmente) werden auf einheitliche Weise dargestellt durch
 < eindeutiger Name, Typ, Repräsentation>.

Bild 19 zeigt ein Objekt vom Typ Dateisystem, das aus einzelnen Dateien besteht. Die Adressierung der Objekte erfolgt über Ca-

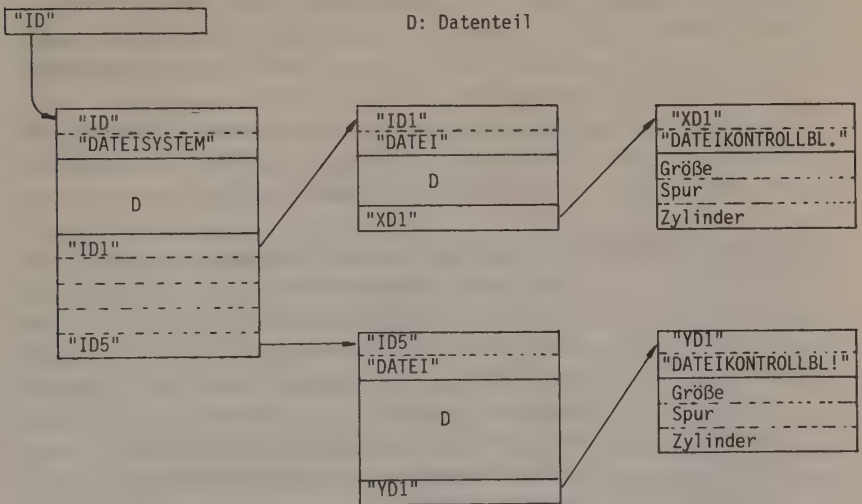


Abb. 19: Capability-Verzeigerung des Datentyps Dateisystem

pabilities, die in Capability-Listen gespeichert sind. Der Repräsentationsteil jedes Objekts enthält einen Datenteil und eine Capability-Liste, deren Capabilities auf Teilobjekte zeigen, aus denen das Objekt zusammengesetzt ist. Damit kann ein Abstraktionsmechanismus zur Verfügung gestellt werden, der die Bildung neuer Datentypen aus bereits bestehenden Typen erlaubt. Dazu sind Funktionen im Betriebssystemkern enthalten. Aus den zusammengesetzten Typen können durch Hinzufügen von Prozeduren zur Manipulation der Komponententypen abstrakte Datentypen gewonnen werden. Objekte eines abstrakten Typs sind nur über die

definierten Prozeduren zugänglich. Capabilities sollen den Zugriff auf Teilkomponenten eines abstrakten Typs ohne die zugehörigen Prozeduren verhindern. Ein Prozeß besteht in HYDRA aus einer Folge von Operationsaufrufen abstrakter Datentypen. Die Schutzumgebung eines Prozesses besteht in den erreichbaren Capabilities und ändert sich bei jedem Operationsaufruf.

Aus den realisierten Konzepten lassen sich wesentliche Teile des Betriebssystemkerns ableiten:

- Die Schutzinformation selbst ist zu schützen, d.h., es müssen Operationen zur Manipulation der Capabilities bereitstehen.
- Der Prozeduraufruf und damit der Wechsel der Schutzumgebungen muß unterstützt werden, da damit ein Wechsel der Programmbe-reiche und der Datenbereiche verbunden ist.
- Die Organisation abstrakter Typen muß unterstützt werden, da damit sowohl die Verwaltung der Capability-Teile als auch die Zugriffsprüfungen auf Objekte verbunden sind.

Folgende Kernoperationen stehen zur Verfügung:

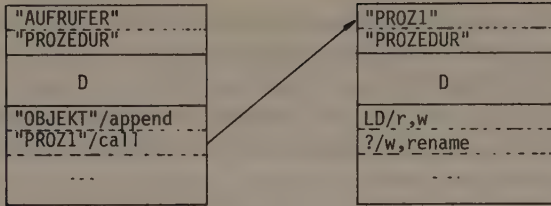
GETDATA	Hole Daten aus Datenteil
PUTDATA	Schreibe Daten in Datenteil
ADDDATA	Hänge Daten an Datenteil
LOAD	Hole Capability aus Capability-Teil
STORE	Speichere Capability in Capability-Teil
APPEND	Hänge Capability an Capability-Teil
DELETE	Lösche Capability aus Capability-Teil
DESTROY	Zerstöre Objekt
COPY	Kopiere Objekte in ein neu anzulegendes Objekt
CREATE	Erzeuge neues Objekt

Diesen Funktionen sind spezielle Rechte im Rechteteil einer Capability zugeordnet. Sie werden um Rechte für spezielle Schutzprobleme erweitert. Der Rest des 24 Bit breiten Rechtevektors steht für Rechte in explizit definierten (abstrakten) Typen zur Verfügung. Ein Beispiel eines speziellen Schutzproblems ist die Weitergabe von Capabilities. Um die unkontrollierte Verteilung im System zu verhindern, wird ein "copy"-Recht eingeführt, das die Weitergabe einer Capability erlaubt, wenn deren Rechtevektor das Recht enthält.

Zur Verdeutlichung des Wechsels von Schutzzumgebungen soll ein Prozeduraufruf genauer beschrieben werden (Bild 20). Eine Prozedur muß nach ihrem Aufruf auf eigene Daten zugreifen, die in ihrer Capability-Liste stehen. Zweitens muß auf Benutzerdaten zugegriffen werden, mit Rechten, die der Benutzer selbst nicht hat, aber die aufgerufene Prozedur. Drittens müssen zum Zugriff auf Objekte bestimmte Rechte in der Capability des Benutzers vorhanden sein, deren Fehlen zum Abbruch der Prozedur führte. Und viertens muß bei Zugriff auf abstrakte Typen möglicherweise auf implementierende Objekte zugegriffen werden, d.h., die Capabilities der implementierenden Objekte müssen erreichbar sein. Dazu werden Schablonen zur Verfügung gestellt. Die Bedingung 1 wird durch die Capability-Liste einer Prozedur ermöglicht. Bedingung 2 wird durch eine Schablone ermöglicht, in der die Rechte stehen, die die Prozedur zur Ausführung braucht. Beim Aufruf wird die Schablone (formaler Parameter) mit der aktuellen Capability vereinigt. Bedingung 3 wird realisiert, indem die aktuelle Capability auf bestimmte Rechte überprüft wird. Bei zusätzlicher Angabe des Typs können auch dynamische Typprüfungen durchgeführt werden. Bedingung 4 schließlich wird durch ein spezielles Recht ermöglicht, das den Zugriff auf implementierende Objekte erlaubt. Die Abb. 20 soll den Umgebungswechsel und die Verwendung von Schablonen bei Prozeduraufrufen verdeutlichen. Das Programm "AUFRUFER" hat Zugriff auf die Prozedur "PROZ1" und das "append"-Recht für "OBJEKT". "PROZ1" hat eine Parameterschablone mit den Rechten "w" und "rename". Nach dem Aufruf hat die Prozedur die Rechte "w" und "rename" für das aktuelle Objekt "OBJEKT".

Der Wechsel der Schutzzumgebungen in HYDRA läßt sich mit einer Schutzmatrix einfach darstellen. Bei jedem Prozeduraufruf entsteht eine neue Zeile der Matrix, die beim Verlassen der Prozedur wieder entfernt wird. Die Schablonen lassen sich durch Matrixzeilen darstellen, in denen einige Rechte und Objektnamen nicht besetzt sind. Sie werden aber nicht in der Matrix geführt, da sie erst beim Aufruf zu den Subjekten gerechnet werden. Bild 21 repräsentiert den Prozeduraufruf aus Bild 20 in einer Schutzmatrix.

Vor dem Aufruf:



Nach dem Aufruf:

"PROZ1"
"PROZEDUR"
D
LD/r,w
"OBJEKT"/w, rename
...

D: Datenteil
 LD: lokale Daten der
 Prozedur "PROZ1"
 r: Leserecht
 w: Schreibrecht

Abb. 20: Schutzumgebungswechsel bei Prozeduraufruf

Nach dem Aufruf existiert eine neue Zeile und eine neue Spalte für die lokalen Daten (LD) der Prozedur. Es werden nur die wesentlichen Zeilen und Spalten der Schutzmatrix dargestellt.

Schutzmatrix		"PROZ1"	"OBJEKT"	
"AUFRUFER"		call	append	

Schutzmatrix		"PROZ1"	"OBJEKT"		LD
"AUFRUFER"		call	append		
"PROZ1"			write rename		read write

LD: lokale Daten der Prozedur "PROZ1"

Abb. 21: Änderungen der Schutzmatrix bei Prozeduraufrufen

Das Betriebssystem PSOS

Im Betriebssystem PSOS ("provably secure operating system", /NEUM77/) wird neben der Integration von Schutzmechanismen auch versucht, einen Sicherheitsbegriff zu definieren und mit Hilfe der Betriebssystemspezifikation nachzuweisen, daß die Spezifikation und die Implementation den definierten Sicherheitsbegriff erfüllen. PSOS erweitert einerseits die Vorgehensweise in ADEPT-50, da nicht nur Zugriffe auf spezielle Objekte kontrolliert werden. Andererseits ist es in die Reihe der Betriebssysteme einzuordnen, die einen definierten Sicherheitsbegriff aufgrund formaler Aussagen über Programme nachweisen. Ein dritter Aspekt bei PSOS ist die Verwendung einer Spezifikationsmethode, in die auch der Nachweis von Sicherheitsaussagen integriert ist (siehe auch 1.3.1).

Der PSOS zugrunde liegende Schutzmechanismus (Capabilities),

die Idee eines Betriebssystemkerns, Möglichkeiten zur Definition abstrakter Datentypen sind analog zu HYDRA und brauchen nicht weiter ausgeführt zu werden. Capabilities werden nicht in Listen verwaltet, sondern durch ein Markierungsbit von anderen Datentypen durch die Adressierungshardware unterschieden. Dies erlaubt eine relativ einfache Kontrolle der Capability-Erzeugung und -Verwendung. Das Rechenwerk wird um Capability-Register erweitert. Dadurch wird einerseits die Prozeß- und die Prozedurverwaltung effizient implementiert. Andererseits läßt sich die Capability-Verwaltung effizient realisieren. Auch die Struktur des Betriebssystems, eine Hierarchie abstrakter Maschinen, wo jeder Ebene eine Nummer ("level") zugeordnet ist, spiegelt sich in Hardware-Maßnahmen wider. In einem Ebenenregister wird die Nummer der augenblicklichen Betriebssystemebene geführt. Es sind nur solche Betriebssystemoperationen aufrufbar, die auf derselben Ebene oder tiefer liegen. Mit dem Prozeduraufruf wird automatisch auch das Ebenenregister angepaßt.

1.2.3 Schutzkonzepte in Programmiersprachen

Schutzaspekte wurden in Programmiersprachen zunächst aus Strukturierungsgesichtspunkten integriert. ALGOL60 unterbindet durch die Blockstruktur u.a. den willkürlichen Gebrauch von Variablen. Dabei werden Gültigkeitsbereich und Existenzbereich von Variablen für den Programmierer transparent und statisch definierbar, die Verwendung von Blöcken wird nicht erzwungen. Auch die Speicherverwaltung eines Programms wird vom Übersetzer organisiert. Die Verwendung von höheren Programmiersprachen an sich ist eine Schutzmaßnahme, da von Maschinendetails in den Programmen abstrahiert werden kann. Dieser Aspekt wird im System/38 ausgenutzt, wo keine Assemblersprache zur Verfügung gestellt wird, sondern die von Übersetzern für höhere Programmiersprachen erzeugten Programme durch einen Mikrobefehl in Mikroprogramme übersetzt werden.

Die Übersetzer höherer Programmiersprachen stellen i.allg. dynamische Überprüfungen, z.B. von Feldgrenzen, zur Verfügung, die aber nicht verwendet werden müssen.

Trotz der statischen Kontrolle der Blockstruktur und der Variablenzugriffe muß der Schutzmechanismus in einer dynamischen Schutzmatrix dargestellt werden, wenn rekursive Prozeduraufrufe erlaubt sind. Jedem Block ist eine Zeile zugeordnet (Subjekt), die Objekte bestehen aus den Variablen. Deren Bezeichner sind für ihre Eindeutigkeit um Blocknummern zu erweitern. Zusätzlich gehören auch Marken und Prozeduren zu den Objekten, da diese in Sprüngen, Prozeduraufrufen oder als aktuelle Parameter auftreten können. Bild 22 zeigt ein kleines Programm und die Darstellung der Zugriffsbeziehungen in einer Schutzmatrix. Bei Prozeduraufrufen müssen, analog zur Vorgehensweise in Abb. 21, weitere Zeilen und Spalten angehängt werden.

Modularisierungsmöglichkeiten in Programmiersprachen erleichtern die Berücksichtigung von Schutzproblemen in Programmen. Statisch wird das Programm übersichtlicher, kleine Programmeinheiten mit wenigen Querbeziehungen unterstützen die Verifizierbarkeit. Zugriffe zwischen verschiedenen Moduln sind bei der Übersetzung kontrollierbar. Dynamisch können Moduln als Schutzumgebungen interpretiert werden, die beim Übergang zwischen Moduln (z.B. beim Prozeduraufruf) angepaßt werden. /LIND76/ stellt die enge Beziehung zwischen abstrakten Datentypen, einem speziellen Modularisierungskonzept, und kleinen Schutzumgebungen ("small domains") heraus. Die Abstraktionseigenschaft wird zur Verbesserung sowohl der statischen als auch der dynamischen Programmstruktur verwendet. Der erste Vertreter dieser Programmiersprachenklasse war SIMULA67 mit dem Klassenkonzept. Sprachen mit ähnlichen Konzepten wurden erst wieder Mitte der siebziger Jahre zur Verbesserung der Programmentwicklung eingeführt. Dazu zählen MODULA, MESA, CLU, EUCLID, Ada und Modula-2. Sie enthalten ein allgemeines Modularisierungskonzept ("MODULE", "PACKAGE"), das das Programmieren mit abstrakten Datentypen (teilweise nur bedingt) unterstützt. Die Allgemeinheit des Modularisierungskonzepts widerspricht teilweise den speziellen Eigenschaften abstrakter Datentypen (siehe Kapitel 3).

	A1	B1	C1	D1	A2	F2	E2	
äußerer Block	r,w	r,w	r,w	r,w				
innerer Block		r,w	r,w	r,w	r,w	r,w	r	

Zugehöriges Programm:

BEGIN

A,B: INTEGER;

B,C: REAL;

...

BEGIN

A,F: INTEGER;

CONST E: BOOLEAN;

...

END INNERER BLOCK;

END AUSSERER BLOCK.

Abb. 22: Die Blockstruktur eines Programms, dargestellt in einer Schutzmatrix

Da das jeweilige Modulkonzept bei der Implementierung von Schutzmaßnahmen zur Verfeinerung der Schutzumgebungen verwendet wird, hat es auf eine Schutzmatrix ähnlichen Einfluß wie das HYDRA-Typkonzept in 1.2.2 und braucht nicht wiederholt zu werden.

Nicht allgemein bekannte programmiersprachliche Schutzkonstrukte

Obwohl es keine speziellen Konstrukte zur Formulierung von Schutzproblemen in bekannten Programmiersprachen gibt, gibt es einige interessante Ansätze, die die folgenden Zielsetzungen gemeinsam haben:

- 1) Unterscheidung zwischen statischen und dynamischen Schutzmaßnahmen
- 2) Unterscheidung zwischen Mechanismus und Strategie
- 3) Unterstützung der Verifikation

Die erste Eigenschaft soll die Effizienz von Schutzmaßnahmen verbessern, da rein dynamische Prüfungen zu langsam und rein statische Maßnahmen zu eingeschränkt sind. Eigenschaft zwei soll dem Programmierer erlauben, spezielle Schutzprobleme mit allgemeinen Konstrukten zu formulieren. Die Konstrukte sollen also einen möglichst großen Anwendungsbereich haben. Die dritte Eigenschaft ergibt sich aus dem engen Zusammenhang zwischen zuverlässiger und sicherer (siehe 1.1.2) Software. Ähnliche Eigenschaften werden auch für Synchronisationskonstrukte gefordert, um dem Programmierer die Formulierung allgemeiner Synchronisationsprobleme und deren Verifikation zu erleichtern. Obwohl es dazu verschiedene Ansätze gibt (Monitore, Rendezvous, Semaphore), konnte noch kein Konsens über ein allgemeines Synchronisationskonstrukt in Programmiersprachen erreicht werden, da die bekannten Ansätze sich jeweils auf die Formulierung eines speziellen Mechanismus beschränken.

Morris versucht in /MORR73/, Schutzprobleme wie Identifikation und Zugriffskontrolle zu formulieren. Dazu benötigt er Lokalität und eindeutige Identifikation von Objekten. Mit Lokalität ist die Beschränkung der Zugriffspfade auf ein Objekt gemeint. Diese Eigenschaft wird z.B. in abstrakten Datentypen erfüllt, da dort ein Objekt von Typ T nur über Operationen O(T) erreichbar ist. Die eindeutige Identifikation realisiert Morris in einem abstrakten Typ mit den Operationen

"verberge" und "enthülle".

Ein mit "verberge" erzeugter Objektname kann nur mit einer zugehörigen "enthülle"-Operation zum Objektzugriff verwendet wer-

den. Die beiden Operationen realisieren damit einen dynamischen Schlüssel-Schloß-Mechanismus, wo jeweils der Besitz einer passenden "enthülle"-Operation den Objektzugriff gestattet.

In /MCGR79/ werden explizite Sprachkonstrukte für einen Capability-Mechanismus in PASCAL eingebettet (die Trennung zwischen Strategie und Mechanismus ist damit nicht gegeben). Durch die Einbettung in PASCAL ist eine Erweiterung der Hoareschen Verifikationsmethode möglich. Die Trennung zwischen statischen und dynamischen Schutzkontrollen erfolgt über zwei Konstrukte: "grant" und "capability".

Mit dem "grant"-Konstrukt können Variablen in Blöcken bekannt gemacht werden, um so die Schutzmöglichkeiten des Blockkonzepts zu verallgemeinern.

grant <Variablenliste> to <Blockliste>

Damit können bei der Übersetzung bereits Zugriffe zwischen Blöcken auf ihre Zulässigkeit überprüft werden. Die Einschränkungen sind in Form einer Schutzmatrix darstellbar, wobei die Blöcke als Subjekte und die Variablen, Prozeduren und Marken als Objekte zu interpretieren sind.

Zur dynamischen Schutzprüfung werden, als Erweiterung der in /JONE76/ vorgeschlagenen Konstrukte, dynamisch erzeugte Objekte über Capabilities angesprochen:

var myfile: file capability;

(Die Variable "myfile" ist vom Typ "file capability", so daß die für den Datentyp "file" definierten Operationen ("read", "write") zulässige Rechte der Capability "myfile" sind). Eine Capability wird entweder bei Generierung eines Objekts erzeugt (analog zum POINTER-Konzept in PASCAL)

myfile := file.create;

oder durch Wertzuweisung einer anderen Capability

yourfile := myfile {read};

(Die Capability-Variable "yourfile" zeigt auf das gleiche Objekt wie "myfile", kann aber nur die Operation "read" ausführen, sofern das Recht im Rechtsteil von "myfile" enthalten ist.)

Dynamische Zugriffskontrollen werden auf gemeinsam benutzte Objekte beschränkt, die unter einem Monitor (Hoare) zusammengefaßt werden. Die Rechtenamen werden durch das Konstrukt "monitor" definiert. Mit der Operation "rights" erhält man die Menge der in der Capability enthaltenen Rechte, mit "object" können zwei Capabilities auf Übereinstimmung des Objektnamens verglichen werden. Bild 23 zeigt das Beispiel des Monitors "file" mit den Operationen "read" und "write". Der Monitor "file" (der Monitor "disk") wird dem Benutzer (dem Monitor "file") statisch zur Verfügung gestellt.

Die Weitergabe einer Capability wird durch das fest vereinbarte Recht "copy" gesteuert. Capability-Zuweisungen (d.h. die Kopie eines Capability-Identifikators) sind nur möglich, falls die Capability der rechten Seite das "copy"-Recht enthält. Eine Capability als aktueller Parameter benötigt kein "copy"-Recht, so daß das Anlegen von Capability-Kopien beim aufgerufenen Monitor verhindert werden kann. Der Monitor kann dem aktuellen Parameter eine Null-Capability zuweisen, so daß beim Aufrufer die übergebene Capability gelöscht wird. Damit können durch Systemroutinen Capabilities zurückgenommen werden ("revocation", Kap.1).

```
System: begin
  type file = monitor ():
    operations read, write;
    procedure read ();
      :
      Disk.read;
      :
    end read;
    :
  end file;
grant file to user;
monitor Disk;
  operations read, write;
  :
  end Disk;
grant Disk {read, write} to file;
process user;
  var my_file: file capability;
  :
  my_file:= file.create(); my_file.read();
  :
  end user;
  :
end System.
```

Abb. 23: Beispielprogramm für die statische und dynamische Capability-Verwendung

1.3 Methoden und Werkzeuge zur Spezifikation und Verifikation von Schutzproblemen

Trotz der genannten implementierten Schutzmaßnahmen sind Schutzprobleme bei der Systementwicklung nur schwer faßbar. Es fehlt einerseits an einheitlichen formalen Hilfsmitteln für verschiedene Schutzprobleme. Wie in Abschnitt 1.2 dargestellt, sind viele der behandelten Probleme mit dem Schutzmatrix- oder dem Verbandsmodell beschreibbar. Ein Vergleich mit Abschnitt 1.1 zeigt aber, daß ein großer Teil der Schutzprobleme nicht berücksichtigt werden kann. Insbesondere die strikte Trennung zwischen Informationsflußmodellen und Zugriffsschutzmodellen und die Verbandsstruktur führen zu starken Einschränkungen bez. der modellierbaren Probleme. Andererseits fehlen einheitliche formale Hilfsmittel für verschiedene Implementationen von Schutzproblemen. Dadurch wird das Verifizieren der Implementation erschwert, selbst wenn das Schutzproblem formal definiert ist.

In /CHEH81/ werden vier formale Spezifikationsmethoden bez. ihrer Anwendbarkeit beim Spezifizieren, Implementieren und Verifizieren von Schutzproblemen verglichen. Bild 24 zeigt eine grobe Übersicht über alle vier Methoden. Die zugehörigen Werkzeuge sind z.T. schlecht aufeinander abgestimmt, die maximale Größe der bearbeitbaren Probleme liegt bei ca. 200-2000 Programmzeilen.

Unabhängig von der Spezifikation verifiziert /POPE78/ ein Betriebssystem bez. eines formal definierten Sicherheitsbegriffs. Interessant daran ist, daß das Betriebssystem (UNIX) nicht um Zugriffsschutzkomponenten erweitert wird, sondern durch geeignete Strukturierung und statische Zugriffseinschränkungen die Sicherheit nachgewiesen wird.

/ANDR80/ schließlich erweitert die Ableitungsregeln der Hoareschen Verifikationsmethode um Regeln, die die Informationsflüsse der Anweisungen eines Programms erfassen. Damit kann der Nachweis von Schutzeinschränkungen in den Korrektheitsbeweis von Programmen integriert werden. Damit sind auch automatische

	HDM	FDM	GYPSY	AFFIRM
entwickelt bei	Stanford Research Institute	System Development Corp.	University of Texas	Univ. of Southern Calif- ISI
zugänglich seit	1979	?	1979	1979
Spezifikationsmethode	operational SPECIAL	algebraisch Ina Jo	operational um Zusicherungen erweitertes PASCAL Codegenerierung für LSI/11	algebraisch
Implementations-sprache	VCG für eine spezielle Zwischensprache, für die ein Übersetzer nach Modula existiert	VCG für eine Modula-Teilmenge		angepaßtes PASCAL mit IMPORT-Klausel, nicht direkt übersetzbar
programmiert in/auf	INTERLISP TOPS-20, TENEX PDP	VM/370 AMDAHL	UCI-LISP TOPS-20 PDP	INTERLISP TOPS-20, TENEX PDP
erreichbar	ARPA-Netz		ARPA-Netz	ARPA-Netz
Anwendungen	KSOS, secure UNIX: Spezifikation auf einer Ebene. PSOS: mehrschichtige Spezifikation, Anwendung eines mehrschichtigen Sicherheitsmodells.	KVM/370: VM/370 mit mehrschichtigen Sicherheitsanforderungen. AUTODINII (sicheres Netzwerk): Verifikation der obersten Ebene.	Kommunikationssysteme mit 1000-2000 Zeilen. Von TI zur Verifikation spezieller Programmteile.	Verifikation von Kommunikationsprotokollen. DELTA-Experiment (Nachrichtensystem): Verifikation von Eigenschaften bez. max. 100 Spezifikations- und 200 Implementationszeilen.

VCG: Bedingungs-generator

TI: Texas Instruments

Abb. 24: Spezifikationsmethoden in einer vergleichenden Übersicht

Verifizierer auf die Verifikation von Sicherheitsaussagen anwendbar.

Nachdem im ersten und zweiten Abschnitt Begriffe und bekannte Implementationsmethoden im Zusammenhang mit Schutzmaßnahmen vorgestellt wurden, sollen im dritten Abschnitt formale Spezifikations- und Verifikationsmethoden und zugehörige Werkzeuge für Schutzprobleme betrachtet werden. Zunächst wird der Spezifikationsbegriff eingeschränkt und die zur Verfügung stehenden Werkzeuge grob klassifiziert. Anschließend wird auf die Metho-

den HDM ("hierarchical development method", entwickelt bei SRI) und FDM ("formal development method", entwickelt bei SDC) eingegangen, da sie Informationsfluß- bzw. Zugriffsschutzmodelle enthalten. Die in /POPE78/ entwickelte Verifikationsmethode wird stellvertretend für statische Schutzuntersuchungen vorgestellt. Schließlich wird auf einen Kalkül zur formalen Ableitung von Informationsflüssaussagen eingegangen.

1.3.1 Spezifikationsmethoden für Schutzprobleme

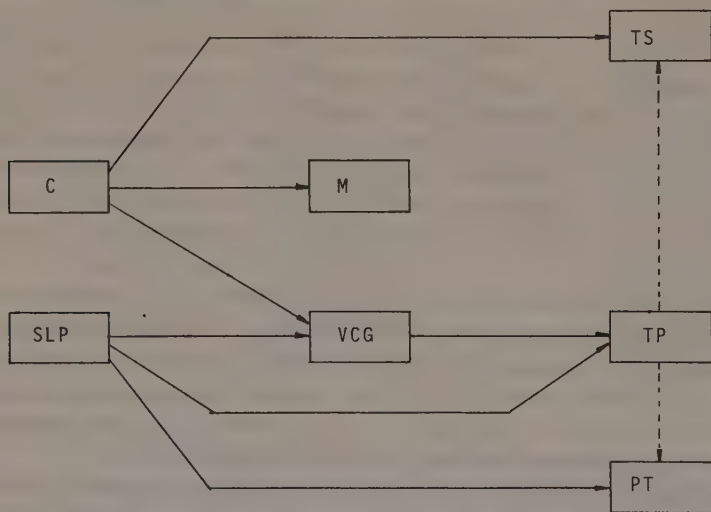
Da der Begriff Spezifikation nicht einheitlich gebraucht wird, soll er im folgenden eingeschränkt werden: Eine Spezifikation ist das Ergebnis des Spezifizierens. I.allg. wird die Festlegung von Programmeigenschaften bei der Anforderungsanalyse und der Programmentwicklung als Spezifizieren bezeichnet. Hier soll nur formales Spezifizieren gemeint sein, also die Beschreibung von Programmeigenschaften in einem formalen Modell.

Automatisch unterstützte Spezifikations-, Implementations- und Verifikationsmethoden enthalten meist folgende Werkzeuge:

- Spezifikationssprachenprozessor ("specification language processor", SLP): Damit soll eine formale Spezifikation syntaktisch analysiert werden. Außerdem werden logische Ausdrücke der Spezifikation für die Verifikation aufbereitet.
- Bedingungszerzeuger ("verification condition generator", VCG): Dieser soll aus den Zusicherungen eines Programms und dem Programm selbst bzw. aus einer formalen Spezifikation zu verifizierende Formeln für den Beweiser ableiten.
- Beweiser ("theorem prover", TP): Dieses Programm soll Formeln auf ihre Korrektheit untersuchen. Dazu müssen Axiome und Sätze eines Kalküls angegeben werden, in dem die Formeln gültig sein sollen.

Bild 25 gibt einen Überblick über die Werkzeuge und ihre gegenseitigen Abhängigkeiten. Ein Pfeil stellt jeweils die Datenflußrichtung zwischen den Werkzeugen dar. Der Vollständigkeit wegen sind noch Compiler (C), Maschine (M), Prototypenzer-

ger ("prototyper", PT) und Testsystem (TS) erwähnt. Die Prototyperzeugung versucht, Spezifikationen in ablauffähige Programme umzuwandeln, die bez. der spezifizierten Eigenschaften äquivalent zur Spezifikation sind. Damit sollen bereits in der Spezifikation Entwurfsfehler und sachliche Mängel erkannt werden. Das Testsystem soll die Validation von Programmen unterstützen. Dabei wird davon ausgegangen, daß Prototyperzeuger, Testsystem und Verifizierer unabhängige Programme sind. Es wäre auch vorstellbar, daß beim Beweisen Informationen für die Prototyperzeugung oder für das Testsystem (Testdaten) abgeleitet werden (gestrichelte Pfeile in Abb.25).



(Erläuterungen der Abkürzungen im Text)

Abb. 25: Zusammenhang zwischen Werkzeugen zum Spezifizieren, Implementieren und Verifizieren

HDM

Die am SRI entwickelte Methode HDM erlaubt die formale Spezifikation von Software-Systemen und unterstützt das Verifizieren einer Spezifikation bez. vorgegebener Kriterien und einer Implementation bez. der zugehörigen Spezifikation. Dazu wird eine Hierarchie abstrakter Maschinen aufgebaut, deren oberste die funktionale Schnittstelle des Software-Systems beschreibt. Jede abstrakte Maschine setzt sich aus den Funktionen eines oder mehrerer Moduln zusammen. Ein Modul enthält die operationelle Spezifikation eines abstrakten Datentyps in der Spezifikationssprache SPECIAL. Darin wird der Zustand eines abstrakten Objekts (Objekts eines abstrakten Typs) durch verborgene (H-) Funktionen ("hidden functions") beschrieben. Mögliche Änderungen des Objekts sind in O-, OV-, V-Funktionen spezifiziert, die entweder Werte von H-Funktionen ändern oder ändern und sichtbar machen oder nur sichtbar machen können. Die Hierarchie der abstrakten Maschinen wird in einer speziellen Hierarchiebeschreibungssprache ("hierarchy specification language", HSL) angegeben. Die Schnittstellen der zu spezifizierenden Hierarchieebenen ("interfaces"), die dabei verwendeten Moduln und Abbildungen zwischen den abstrakten Maschinen sind darin zu definieren. Die Abbildungen ("mappings") bilden jeweils die abstrakte Maschine der Ebene i auf die abstrakte Maschine der Ebene i-1 ab. Bild 26 gibt eine Übersicht über die Spezifikations- und Implementationselemente und Eingriffspunkte der zur Verfügung stehenden Werkzeuge.

Die Modulspezifikation und die Hierarchiebeschreibung werden mit dem SLP-Programm analysiert. Das VCG-Programm erzeugt aus der Systemhierarchie vom Beweiser zu verifizierende Bedingungen, die auch zum Korrektheitsbeweis der Implementationen als Vor- und Nachbedingungen verwendet werden. Die oberste abstrakte Maschine, die die funktionale Schnittstelle des Systems repräsentiert, kann ebenfalls bez. eines vorzugebenden Modells verifiziert werden. Die Implementationen der einzelnen Moduln müssen von eigenen sprachabhängigen Verifikationsprogrammen bearbeitet werden. Schließlich werden die Implementationen gemeinsam oder einzeln, abhängig vom Übersetzer, übersetzt und zu

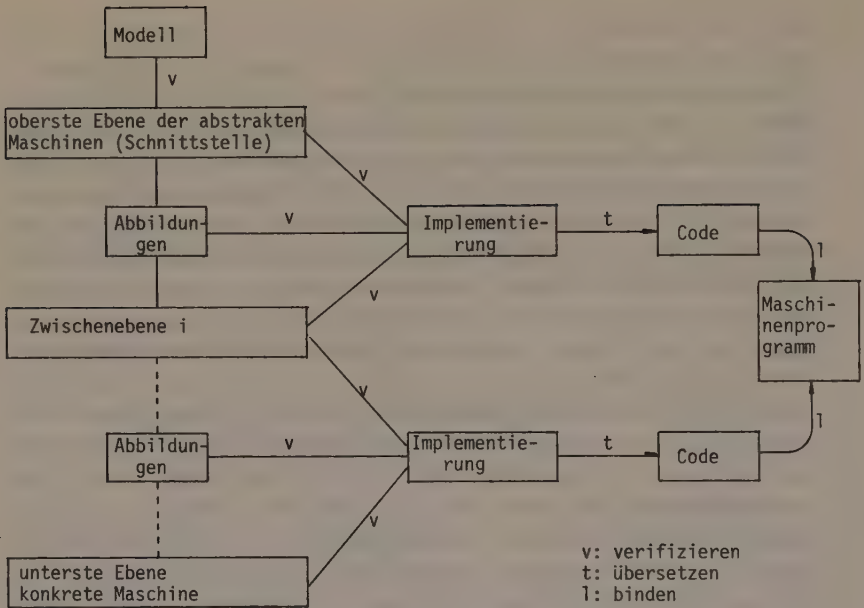


Abb. 26: Schnittstellen in HDM

einem lauffähigen Programm gebunden.

Die Vorgehensweise zur Verifikation der Sicherheitsanforderungen ist aus Abb. 27 ersichtlich. Die funktionale Schnittstelle des Systems kann auf die Einhaltung des mehrschichtigen Sicherheitsmodells ("multilevel security") überprüft werden. Als Werkzeug steht ein Formelgenerator für das Sicherheitsmodell zur Verfügung ("multilevel security formula generator", MLS). Das mehrschichtige Sicherheitsmodell entspricht dem Verbandsmodell von Denning und hat folgende Eigenschaften:

- Leseoperationen eines Prozesses der Klasse i auf ein Objekt der Klasse j sind erlaubt, wenn $i \geq j$ (einfache Sicherheitsbedingung).
- Schreiboperationen eines Prozesses der Klasse i auf ein Objekt der Klasse j sind erlaubt, wenn $i \leq j$ (*-Bedingung).

Durch die genannten zwei Eigenschaften wird die Informationsfluß-Relation " $\rightarrow$ " des Verbandsmodells definiert. Zur Generie-

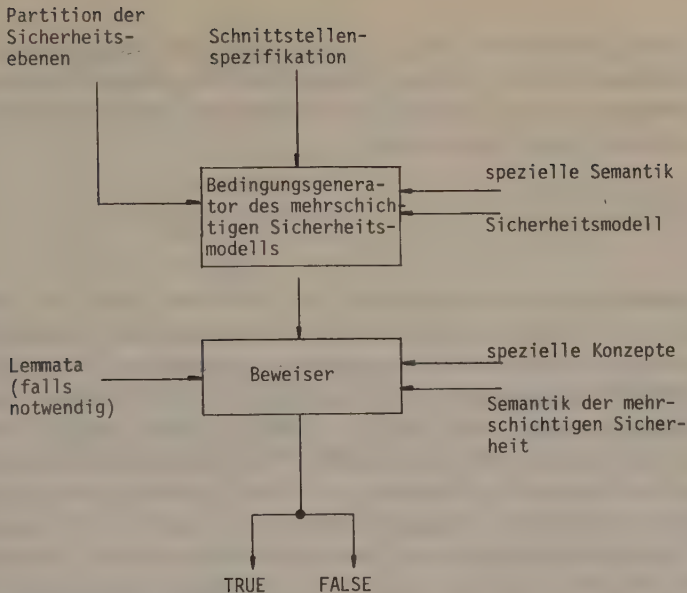


Abb. 27: Verifikation von Sicherheitseigenschaften in HDM

nung der Verifikationsformeln müssen die fehlenden Elemente des Verbands, die Grundmenge und der lub-Operator, definiert werden. SPECIAL selbst enthält keine entsprechenden Sprachkonstrukte, sondern diese werden von MLS bei der Formelgenerierung interaktiv erfragt (Bild 27, Partition der Sicherheitsebenen). Zusätzlich wurden zur Analyse versteckter Kanäle spezielle Eigenschaften der Spezifikationsmethode in den Formelgenerator integriert. Die generierten Formeln werden mit Hilfe des Beweisers, der durch Interaktionsmöglichkeiten vom Benutzer bei Ableitungsschritten unterstützt werden kann, nachgewiesen oder widerlegt.

Obwohl Informationsflußprobleme in HDM automatisch unterstützt werden, bleibt die Einschränkung der Hilfsmittel auf das Mehrebenensicherheitsmodell. Schutzprobleme, die nicht in diesem Modell beschreibbar sind, sind ohne automatische Unterstützung zu verifizieren. Ebenso sind Sicherheitsstrategien, die sich nicht auf die mehrschichtige Sicherheit zurückführen las-

sen, nicht automatisch verifizierbar. Durch die Integration der Sicherheitsstrategie in das Modell und MLS bleiben wichtige Eigenschaften in der Spezifikation verborgen, die zum besseren Verständnis sichtbar sein müßten.

FDM

Die Methode FDM unterscheidet sich von HDM sowohl in Bezug auf die Spezifikation und die Hierarchisierung des Systems als auch durch die axiomatische Spezifikationsmethode. Jede Ebene der Hierarchie kann als Zustandsmaschine aufgefaßt werden, wobei alle Maschinen dasselbe System in verschiedenen Abstraktionsstufen beschreiben. Die oberste Ebene kann als Anforderungsspezifikation interpretiert werden, da dort Korrektheitskriterien möglichst problemnah (abstrakt) formuliert werden. Die notwendigen Verifizierungsschritte leiten sich aus diesen Kriterien ab. Die tiefer liegenden Ebenen verfeinern die abstrakte Darstellung, so daß die unterste Ebene als Spezifikation der Implementation angesehen werden kann, die stark mit dem Programmcode korrespondiert. Zwischen zwei aufeinanderfolgenden Ebenen werden Abbildungen definiert, die die Elemente der unteren auf die Elemente der oberen Ebene abbilden. Im Unterschied zu HDM werden dabei nicht nur die Datenstrukturen (H-Funktionen) ineinander überführt. Bild 28 zeigt die FDM-Struktur eines Systems. Durch Kriterien ("criterion") werden die erlaubten Maschinenzustände definiert und durch Einschränkungen ("constraint") die Zustandsübergänge. Diesbezüglich kann jede Maschinenebene für sich verifiziert werden. Dabei erzeugt der Ina-Jo-Prozessor (Ina-Jo ist die verwendete Spezifikationssprache) aus den Spezifikationen Formeln für den Beweiser. Bei neu hinzugekommenen Transformationen unterer Ebenen müssen die Einschränkungen dafür aus den jeweils darüberliegenden Ebenen abgeleitet werden. Auf unterster Ebene müssen aus der Spezifikation Vor- und Nachbedingungen der Implementationsprogramme erzeugt werden, die weiter im VCG bei Programmverifikation verwendet werden

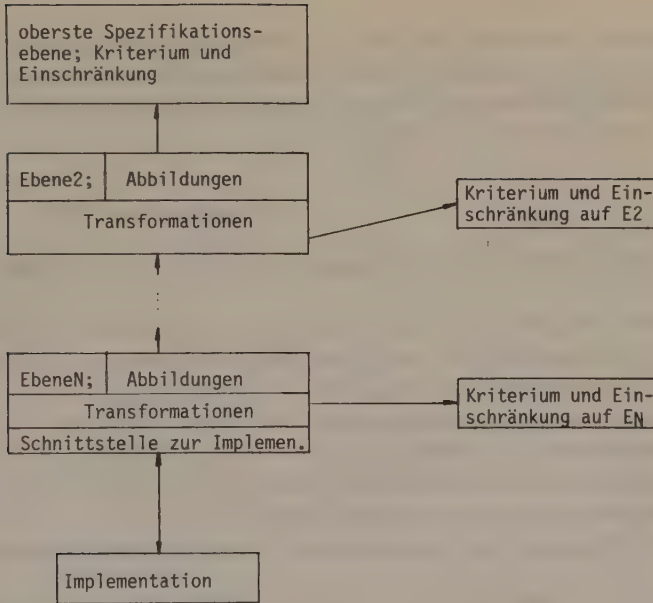


Abb. 28: Systemstruktur in FDM

Zur Verifikation von Sicherheitsstrategien sind in FDM keine speziellen Werkzeuge (wie MLS) vorgesehen. Vielmehr sind die Konstrukte "criterion" und "constraint" zur Spezifikation von Schutzproblemen zu verwenden. Damit ist die Einschränkung der Schutzprobleme auf Zugriffsschutzprobleme verbunden, da der Formelgenerator VCG keine Informationsflußanalyse der Spezifikation durchführt.

Wie HDM enthält auch FDM keine Sprachkonstrukte zur Spezifikation von Schutzeinschränkungen. Auch bei der Verifikation sind keine speziellen Werkzeuge vorgesehen. Die formulierbaren Schutzprobleme unterscheiden sich syntaktisch nicht von anderen Problemen und sind auf Zugriffsschutzprobleme beschränkt. Beide Methoden enthalten keine eigenen sprachlichen Konstrukte zur Formulierung von Schutz- und Synchronisationseigenschaften. Gerade dieser Punkt scheint für eine korrekte Implementation wichtig, da das Fehlen solcher Konstrukte zu Fehlern bei der Problemspezifikation führen kann, insbesondere, wenn asynchrone

und/oder geschützte Systeme zu entwickeln sind. Ähnlich führte die Entfernung von Bedingungs- und Schleifenkonstrukten in Programmiersprachen und ihre Umschreibung mit Sprüngen und Marken zu ständigen Fehlerquellen.

1.3.2 Verifikation von Schutzeigenschaften in Programmen

Popek berichtet in /POPE78/ über die Verifikation eines an der UCLA entwickelten sicheren Betriebssystemkerns ("UCLA-kernel"). Dazu waren folgende Schritte notwendig:

- Definition der Kernfunktionen des Betriebssystems
- Modellierung des Kerns
- Definition eines Sicherheitsbegriffs im Modell
- Programmierung der Kernfunktionen
- Verifikation der Kernfunktionen bez. des Sicherheitsbegriffs

Der Betriebssystemkern enthält Funktionen zur Verwaltung der Schutzzinformationen. Diese sind realisiert durch Capability-Listen, die jedem Prozeß zugeordnet sind. Die Rechemenge einer Capability ist für ein konkretes System fest vorgegeben, aber im Konzept variabel. Außerdem stehen Funktionen zur Verwaltung des Externspeichers und des segmentierten virtuellen Speichers zur Verfügung. Schließlich sind Funktionen zur Prozeßverwaltung und zum E/A-Verkehr im Kern enthalten. Diese Funktionen sind als Kernaufrufe ("supervisorcall", SVC) für Benutzerprozesse sichtbar, wobei die Funktionen zur Veränderung der Capability-Menge oder zur Veränderung der zugeteilten Betriebsmittel nur über zwei spezielle Prozesse zugänglich sind. Die Korrektheit dieser Prozesse ist ebenfalls für die Einhaltung der Sicherheit wichtig, während sonst die Kernfunktionen nicht von den aufrufenden Prozessen abhängen. Bild 29 zeigt grob die Aufteilung des Betriebssystems in Kernoperationen, spezielle Prozesse und Benutzerprozesse.

Die Modellierung des Kerns erfolgt mit Hilfe einer Zustandsmaschine, deren Übergänge durch die Kernoperationen definiert sind. Jeder Zustand der Maschine ist charakterisiert

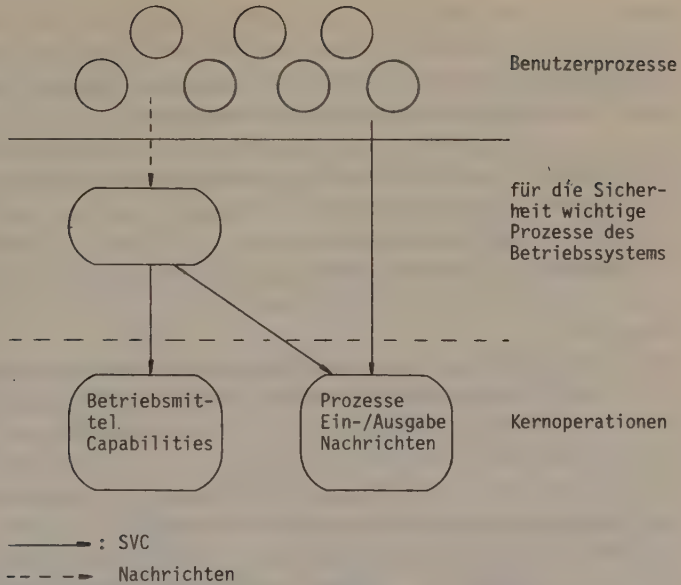


Abb. 29: Zugriffsmöglichkeiten auf Kernoperationen

durch eine Menge von Objekten, den E/A-Aufträgen und einem laufenden Prozeß. Die Operationen werden als unteilbar vorausgesetzt, so daß bei der Verifikation asynchrone Eigenschaften nicht berücksichtigt werden müssen. Das Modell verlangt die Kenntnis aller Objekte, so daß keine Zerstör- oder Erzeugeoperationen berücksichtigt werden müssen. Die Typen der Objekte müssen ebenfalls bekannt sein, da sie die Ausführung von Operationen beeinflussen. Wichtigste Voraussetzung ist die Eindeutigkeit der Objektnamen, die durch folgende Bedingung charakterisiert wird:

S: Zustandsmenge

$$(\forall s \in S)(\forall o_1, o_2 \in s.objects) o_1.name = o_2.name \text{ ---} \rightarrow o_1 = o_2$$

(Wenn die Namen zweier Objekte übereinstimmen, müssen sämtliche Komponenten (Typ, Speicherplatz, Wert) übereinstimmen.)

Der Objekttyp "capability" enthält drei Komponenten.

$c = \langle process_name, object_name, access_type \rangle$

"process_name" bezeichnet das Subjekt, dem die Capability zuge-

ordnet ist, "object_name" das durch die Capability bezeichnete Objekt und "access_type" die zugeteilten Rechte. In der Implementation entfällt die Komponente "process_name", da sie implizit durch die Capability-Liste gegeben ist.

Zur Definition des Sicherheitsbegriffs werden die Relationen

- gewollter lesender Zugriff
- gewollter schreibender Zugriff
- tatsächlicher lesender Zugriff
- tatsächlicher schreibender Zugriff

eingeführt.

- 1) Ein Befehl i will in einem Zustand s auf das Objekt n lesend (schreibend) zugreifen, wenn der Befehl i das Objekt n in seiner Parameterliste als zu lesendes (schreibendes) Objekt enthält.
- 2) Ein Befehl i ändert in einem Zustand s das Objekt o tatsächlich, wenn für den Nachfolgezustand s' unter i von s kein Objekt o' identisch mit o ist.
- 3) Der tatsächliche lesende Zugriff wird durch seine Negation ausgedrückt. Der Befehl i greift auf das Objekt o im Zustand s_1 nicht lesend zu, wenn die Auswirkungen des Befehls i auf alle von o verschiedenen Objekte nicht von o abhängen, d.h., Änderungen von o keinen Einfluß auf von o verschiedene Objekte haben.

Mit Hilfe der genannten Relationen definiert Popek drei Sicherheitsbedingungen, die von den implementierten Operationen erfüllt werden müssen. Dazu schränkt er die Menge aller Objekte auf die Menge der zu schützenden Objekte ein, da nicht alle Objekte für die Sicherheit des Systems wichtig sind.

- 1) Geschützte Objekte können nur tatsächlich geändert werden, wenn sie auch gewollt geändert werden.
- 2) Auf geschützte Objekte kann nur tatsächlich lesend zugegriffen werden, wenn auf sie gewollt lesend zugegriffen wird.
- 3) Ein spezieller Zugriff eines Prozesses auf ein geschütztes Objekt o ist nur erlaubt, wenn das dem Zugriff zugeordnete Recht (z.B. "lesen") in der Rechtemenge der Capability $c=(z,o,access_type)$ enthalten ist.

Zur Implementierung der Kernfunktionen wurde die Programmiersprache PASCAL verwendet, da dafür ein halbautomatischer Verifizierer zur Verfügung stand. Für jede Kernfunktion sind Vor- und Nachbedingungen aus den genannten Sicherheitsbedingungen, der Eindeutigkeitseigenschaft und den eigentlichen Operationseffekten und der Konkretisierung des Modells abzuleiten. Ebenso müssen die Ein- und Ausgabeparameter der Operationen definiert sein, um Sicherheitsbedingung 1 und 2 überprüfen zu können.

Die vorgeschlagene Methode ermöglicht eine einfache Überprüfung der vorgegebenen Sicherheitseinschränkungen, macht aber einige wichtige Annahmen:

- Die Anzahl der Objekte, die Anzahl der Objekttypen und die Menge der Rechte ist fest vorzugeben.
- Es wird davon ausgegangen, daß das Vollständigkeitspostulat erfüllt ist, das verlangt, daß alle sichtbaren Objekte im Modell erfaßt sind.
- Nebenwirkungen der PASCAL-Programme werden ausgeschlossen und die Verwendung von POINTER-Typen wird nicht berücksichtigt. Da nur Sicherheitsbedingung 3 dynamisch überprüft wird, führt die Entscheidbarkeit der Sicherheitsbedingungen 1 und 2 zu Einschränkungen bei der Programmierung.

Einschränkungen des Modells sind, daß Informationsflüsse nicht untersucht werden und asynchrone Prozesse unberücksichtigt bleiben. Dadurch, daß das Modell nicht in eine allgemeine Spezifikationsmethode integriert ist, ergeben sich Probleme bei der Änderung oder Anpassung des Kerns. Trotz der Allgemeinheit der Sicherheitsbedingungen sind Sicherheitsstrategien getrennt zu implementieren (spezieller Prozeß) und zu verifizieren.

Verifikationsmodell mit Informationsflußanalyse

Andrews stellt in /ANDR80/ (siehe auch /REIT79/) eine Ergänzung des Hoareschen Kalküls zur Informationsflußanalyse in Programmen vor. Der Grundgedanke ähnelt den besprochenen Modellen von Jones/Lipton und Denning (1.1.3), die auch in der statischen Programmanalyse, z.B. als Teil eines Compilers, verwendet werden können. In Abhängigkeit von den berücksichtigten Informationsträgern sind auch versteckte Kanäle auffindbar.

Zur Kontrolle des Informationsflusses wird jeder Variablen x eine Klassifikation x zugeordnet. Die Menge der Klassifikationen bildet einen Verband. low bezeichnet das kleinste und high das größte Element des Verbands. Ein bekanntes Klassifikationschema ist die bereits vorgestellte militärische Klassifikation (1.1.4). Dabei gilt: low=unbeschränkt; high=streng geheim.

Analog zum Hoareschen Kalkül müssen für alle programmiersprachlichen Konstrukte Ableitungsregeln angegeben werden. Diese sollen zunächst als Ergänzung von Antezedens und Sukzedens im Hoareschen Kalkül angesehen werden. Z.B. gilt für die Wertzuweisung (Regel siehe Abb.30): local und global sind Klassifikationsvariablen, die sich aus lokalen und globalen Informationsflüssen ergeben. Ein lokaler Informationsfluß entsteht z.B. bei einer Bedingung, da der durchlaufene Zweig Informationen über die in der Bedingung vorkommenden Variablen enthalten kann. Globaler Informationsfluß entsteht bei Schleifen, da die Terminierung über die Schleifenbedingung Auskunft gibt. Während lokale Informationsflüsse u.U. auf Programmteile beschränkt werden können, haben globale Informationsflüsse Auswirkungen auf die Nachbedingungen des Gesamtprogramms.

Die von einem Programm zu erfüllende Schutzstrategie muß in Ergänzung der Vor- und Nachbedingungen des Programms formuliert werden. Dazu werden Bedingungen über im Programm verwendete Variablen und ihre Klassifikationen vorgegeben. Für ein Sortierprogramm, das ein Feld A mit den Feldgrenzen von 1 bis n sortiert, unter Verwendung der Hilfsvariablen "first", "maxindex", "i" und "temp", könnte die Strategie folgendermaßen defi-

Ergänzter Kalkül

Ableitungsregeln für Wertzuweisung und einseitige Alternative:

$\{P[x \leftarrow \underline{E} \text{ } \textcircled{\text{local}} \text{ } \textcircled{\text{global}}]\} x := E \{P\}$	P Prädikat
$\{V, L', G\} S \{V', L', G'\}$ $V, L, G \vdash L'[\underline{\text{local}} \leftarrow \underline{\text{local}} \text{ } \textcircled{B}]$	V, L, G sind Zusicherungen über die Klassen der Programmvariablen (V) und die der Hilfsvariablen local und global.
$\{V, L, G\} \text{ IF } B \text{ THEN } S \text{ END}; \{V', L, G'\}$	

Ableitung für "A1, A2":

$\{z \leq Z_0 \wedge y \leq y \wedge \underline{\text{local}} = b\} y := x \{z \leq Z_0 \wedge y \leq x \textcircled{b}\}$	$\{z \leq Z_0 \textcircled{\text{local}} \wedge y \leq x \textcircled{b} \wedge \underline{\text{local}} = b\} z := y \{z \leq Z_0 \textcircled{\text{local}} \wedge y \leq x \textcircled{b}\}$
$\{z \leq Z_0\} \text{ IF } b \text{ THEN } y := x \text{ END};$ $\{z \leq Z_0 \wedge y \leq x \textcircled{b}\}$	$\{z \leq Z_0 \wedge y \leq x \textcircled{b}\} \text{ IF } b \text{ THEN } z := y \text{ END};$ $\{(z \leq Z_0 \textcircled{\text{local}} \wedge y \leq x \textcircled{b}) \leftrightarrow z \leq Z_0 \textcircled{\text{local}} \wedge y \leq x \textcircled{b}\}$
$\{z \leq Z_0\} \text{ IF } b \text{ THEN } y := x \text{ END}; \text{ IF } b \text{ THEN } z := y \text{ END}; \{z \leq Z_0 \textcircled{\text{local}} \wedge y \leq x \textcircled{b}\}$	

Abb. 30: Ableitung im ergänzten Kalkül

niert sein (c sei eine beliebige vorgegebene Klassifikation):

$$\{A \leq c\} \text{ SORT } \{A \leq c \textcircled{n}, \text{ temp} \leq n, \text{ maxindex} \leq n, i \leq n\}$$

Das bedeutet, daß die Klassifikation von A nach dem Sortiervorgang nur von der Klassifikation der Obergrenze n und von A abhängt und in den Hilfsvariablen "maxindex", "temp" und "i" keine Information über A gespeichert wird.

Die Ergänzung der Hoareschen Verifikationsregeln führt zu stärkeren Informationsflußeinschränkungen als nötig (Im Sinn des Modells von Jones (1.1.3) ist der so realisierte Schutzmechanismus nicht maximal.). Durch eine Erweiterung der Hoareschen Prämissen läßt sich die Analyse verbessern (vergrößern im Sinn von Jones). Zur Veranschaulichung diene folgendes Beispiel:

A1: if b then y:=x end;

A2: if not b then z:=y end;

(b hängt nicht von x, y oder z ab)

z ist unabhängig von x. Abb. 30 und Abb. 31 zeigen die Ableitungsschritte für das Programm im erweiterten und im ergänzten Kalkül (die verwendeten Ableitungsregeln sind ebenfalls angegeben). Dabei wird im ergänzten Kalkül die Variable z als abhängig von x erkannt, während der erweiterte Kalkül die Unabhängigkeit richtig erkennt, was auch aus der möglichen Umformung von A1; A2 deutlich wird:

A3: if b then y:=x else z:=y end;

Erweiterter Kalkül

Ableitungsregeln für Wertzuweisung und einseitige Alternative:

$\{P[x \leftarrow E, \underline{x} \leftarrow \underline{E} \text{ @ local @ global}]\} x := E \{P\}$

$\{V \Delta B, L', G'\} S \{V', L', G'\}$

$V, L, G \vdash L'[\underline{\text{local}} \leftarrow \underline{\text{local}} \text{ @ } \underline{B}]$

V, L, G siehe

ergänzter Kalkül

$\{V, L, G\} \text{ IF } B \text{ THEN } S \text{ END}; \{V', L, G'\}$

Ableitung für "A1; A2":

$\{b \rightarrow \underline{z} \leq \underline{Z}_0 \wedge \underline{y} \leq \underline{y} \wedge \underline{\text{local}} = \underline{b}\} y := x$
 $\{b \rightarrow \underline{z} \leq \underline{Z}_0 \wedge \underline{y} \leq \underline{x} \text{ @ } \underline{b}\}$

$\{b \rightarrow \underline{z} \leq \underline{Z}_0 \wedge \underline{\text{local}} = \underline{b}\} z := y \{b \rightarrow \underline{z} \leq \underline{Z}_0 \text{ @ } \underline{y} \text{ @ } \underline{b}\}$

$\{\underline{z} \leq \underline{Z}_0\} \text{ IF } b \text{ THEN } y := x \text{ END};$
 $\{b \rightarrow \underline{z} \leq \underline{Z}_0 \wedge \underline{y} \leq \underline{x} \text{ @ } \underline{b}\}$

$\{b \rightarrow \underline{z} \leq \underline{Z}_0 \wedge \underline{y} \leq \underline{x} \text{ @ } \underline{b}\} \text{ IF } b \text{ THEN } z := y \text{ END};$
 $\{b \rightarrow \underline{z} \leq \underline{Z}_0 \wedge \underline{y} \leq \underline{x} \text{ @ } \underline{b}, \underline{b} \rightarrow \underline{z} \leq \underline{Z}_0 \text{ @ } \underline{y} \text{ @ } \underline{b}\}$

$\{\underline{z} \leq \underline{Z}_0\} \text{ IF } b \text{ THEN } y := x \text{ END}; \text{ IF } b \text{ THEN } z := y \text{ END}; \{b \rightarrow \underline{z} \leq \underline{Z}_0 \wedge \underline{y} \leq \underline{x} \text{ @ } \underline{b}, \underline{b} \rightarrow \underline{z} \leq \underline{Z}_0 \text{ @ } \underline{y} \text{ @ } \underline{b}\}$

Abb. 31: Ableitung im erweiterten Kalkül

/ANDR80/ gibt auch Ableitungsregeln für Koprozesse und die Semaphoroperationen "wait" und "signal" an, so daß im Kalkül auch Informationsflüsse über asynchrone Abläufe erkannt werden können.

Der angegebene Kalkül baut auf dem Verbandsmodell auf und ist deshalb nicht allgemein. Durch die statische Untersuchung der Programme können nur mögliche Informationsflüsse erkannt werden, die im dynamischen Ablauf nicht auftreten müssen. Deshalb ist es sinnvoll, die statischen um dynamische Analysen zu ergänzen. Durch die Einbettung in den Hoareschen Kalkül ist es aber möglich, mit Verifizierern und halbautomatischen Beweisern die Informationsflußanalyse durchzuführen und an kritischen Stellen durch Benutzereingaben die automatische Bearbeitung zu unterstützen.

1.4 Anforderungen

Nachdem bekannte Möglichkeiten bei der Spezifizierung, Implementierung und Verifizierung geschützter Systeme vorgestellt wurden, sollen aus den kritischen Anmerkungen Anforderungen an sprachliche und technische Hilfsmittel für geschützte Systeme abgeleitet werden.

Zum Spezifizieren von Schutzproblemen müssen problemunabhängige Sprachkonstrukte entwickelt werden, die es gestatten, beliebige Schutzmaßnahmen zu formulieren. Dadurch sollen bereits beim Spezifizieren Schutzaspekte berücksichtigt werden. Die Integration in eine Spezifikationssprache ist notwendig, um Schutzprobleme dort betrachten zu können, wo sie auftreten. Schutzprobleme hängen vom umgebenden Problembereich ab und dürfen deshalb nicht isoliert untersucht werden. Die Spezifikationsmethode muß auch asynchrone Systeme beschreiben können, wie der Zusammenhang zwischen Synchronisations- und Schutzproblemen zeigt. Die Schutzkonstrukte der Sprache sind nur im Zusammenhang mit einem formalen Modell sinnvoll. Damit ist ihre Semantik eindeutig definierbar und die Formulierung der Probleme ist einheitlich möglich. Die Konstrukte müssen insbesondere von konkreten Mechanismen abstrahieren, um nicht auf bestimmte Schutzprobleme eingeschränkt zu sein.

Auch programmiersprachliche Konstrukte zur Beschreibung von Schutzmaßnahmen müssen unabhängig von konkreten Mechanismen sein. Sie sollten im gleichen formalen Modell wie die Spezifikationskonstrukte definiert werden, um einen einheitlichen Übergang von der Spezifikation zur Implementation zu erreichen.

Die Verifikation von Schutzmaßnahmen muß ebenfalls auf ein Schutzmodell Bezug nehmen. Um die Verifikation der Schutzprobleme nicht von der Verifikation der Programme zu trennen, sollte dasselbe Modell wie bei Spezifikation und Implementation verwendet werden.

2 Das Informationsflußprädikat

Die in Kapitel 1 vorgestellten Ansätze zur Berücksichtigung von Schutzaspekten in Hardware- und Software-Systemen weisen folgende gemeinsamen Merkmale auf:

- Für jede Anwendung wird ein eigenes Modell definiert, in dem die spezifischen Schutzprobleme beschreibbar sind.
- Die Schutzmodelle sind auf spezielle Schutzstrategien (z.B. militärisches Sicherheitsmodell) oder Mechanismen (z.B. Capability-Mechanismus) zugeschnitten.
- Zum umgebenden Hardware- oder Software-System besteht kein direkter Zusammenhang, so daß die Korrektheit der Implementation des Schutzmodells nur schwer nachweisbar ist.
- Die Schutzmodelle sind weder in einer eventuell vorhandenen Spezifikationssprache noch in der verwendeten Programmiersprache repräsentiert.

Da ein einheitliches Modell fehlt, sind Lösungsmöglichkeiten für Schutzprobleme kaum übertragbar. Da die Modelle häufig auf bestimmte Schutzprobleme zugeschnitten sind, sind Änderungen der Probleme nur mit Aufwand zu berücksichtigen. Obwohl Schutzprobleme ursprünglich gemeinsam mit Synchronisationsproblemen entstanden sind und in Kapitel 1 Synchronisationsprobleme als spezielle Schutzprobleme eingeordnet werden, werden beide Problembereiche mit unterschiedlichen Modellen und Methoden behandelt.

Viele der Merkmale ließen sich dadurch ändern, daß eine Spezifikationssprache und eine Implementationssprache zur Verfügung gestellt werden, die auf einem einheitlichen Modell beruhen. Das Modell sollte allgemein genug zur Beschreibung sowohl synchroner als auch asynchroner Aktivitäten sein und sowohl isolierte als auch konkurrierend genutzte Objekte enthalten. Die Spezifikations- bzw. Implementationssprache sollte Konstrukte enthalten, die die Modelleigenschaften exakt repräsentieren. Die möglichen Objektzugriffe sollten bez. Synchronisation und Schutz beliebig definierbar sein.

Zur Erfüllung dieser Anforderungen wird die Betriebssystemmaschine (Keramidis, siehe /KERA82/) als allgemeines Modell ver-

wendet. Die Betriebssystemmaschine ist geeignet zur Modellierung asynchroner, verteilter Prozeßsysteme; synchrone oder sequentielle Prozeßsysteme sind als Spezialfälle enthalten. Die im Modell definierten Ausführungsprädikate für Operationen führen bei Nichterfüllung zur Blockierung der Prozesse. Dabei fällt auf, daß in Monoprozeßsystemen (sequentiellen Systemen) die Blockierung unendlich lange dauert, da keine weiteren Prozesse zur Veränderung der Blockierungszustände vorhanden sind. In diesem Fall verhält sich eine Synchronisationsbedingung wie eine Konsistenzbedingung. Ähnliches gilt für Zugriffsschutzbedingungen. In Abhängigkeit von der verfolgten Schutzstrategie kann der Zugriff eines Prozesses ganz unterbunden oder bis zur Erfüllung einer Bedingung (z.B. die nachträgliche Zuteilung eines Zugriffsrechts) blockiert werden.

Neben reinen Zugriffsschutzproblemen sind auch Informationsflußprobleme modellierbar. Dazu werden die bei Keramidis eingeführten Relationen "semantische Abfrage" und "semantische Änderung" verwendet. Da Informationen ohne Informationsträger nicht existieren, muß es auch zur Informationsübertragung sichtbare Träger geben, auf die zugegriffen werden kann. Das bedeutet, daß Informationsflußprobleme und Zugriffsschutzprobleme nicht mit verschiedenen Modellen beschrieben werden dürfen. Es wird sich zeigen, daß Zugriffsschutzprobleme als Spezialfall von Informationsflußproblemen aufgefaßt werden können.

Das zweite Kapitel führt zunächst das Modell der Betriebssystemmaschine ein. Eine ausführliche Beschreibung und weitergehende Erläuterungen sind bei Keramidis zu finden. Darauf aufbauend wird der Informationsflußbegriff allgemein definiert, und es werden Zusammenhänge zu Synchronisationsproblemen aufgezeigt. Die Definition von Zugriffen führt zur einheitlichen Behandlung von Zugriffsschutz- und Informationsflußproblemen. Schließlich sollen Querbeziehungen zu den in Kapitel 1 beschriebenen Schutzmodellen untersucht werden.

2.1 Das Modell der Betriebssystemmaschine

Die Betriebssystemmaschine baut auf dem Begriff des sequentiellen Prozesses auf. Sie besteht im wesentlichen aus mehreren sequentiellen Prozessen, die über gemeinsame Daten kommunizieren. Zur Formulierung von Synchronisationseigenschaften werden den Prozessen Zustände zugeordnet. Die Zustandswechsel der Prozesse erfolgen aufgrund der Synchronisationsstrategie der Betriebssystemmaschine und der Ausführbarkeitsbedingungen einzelner Prozeßschritte. Die Synchronisationsstrategie wird mit Hilfe einer Prioritäts- und Verträglichkeitsrelation beschrieben, so daß keine Einschränkungen bez. der definierbaren Strategien entstehen.

Sequentieller Prozeß

Ein sequentieller Prozeß wird als Zustandsmaschine durch folgendes 5-Tupel repräsentiert: $P=(A,D,R,a_0,D_0)$ mit

A: Menge von Adressen

D: Datenmenge

$R \subset (A \times D) \times (A \times D)$: Prozeßrelation

$a_0 \in A$: Anfangsadresse

$D_0 \subset D$: initiale Datenmenge

Die Adreß- und Datenmengen A und D bilden die Grundmenge der Prozeßrelation R, die die möglichen Zustandsübergänge des Prozesses P relational definiert. $Q=(A \times D)$ heißt Zustandsmenge des Prozesses P und $q=(a,d) \in Q$ ein Zustand. $q_0=(a_0,d_0)$, $d_0 \in D_0$ bezeichnet einen Anfangszustand des Prozesses P.

Der Berechnungsbegriff definiert die Semantik des sequentiellen Prozesses P. $\langle q_i \mid 0 \leq i \leq m \rangle$ heißt Berechnung des Prozesses P, wenn

$$m \in \mathbb{N}_0 \wedge (\forall i \in \{0, \dots, m\}) (q_i, q_{i+1}) \in R$$

Anstelle von $(q_i, q_{i+1}) \in R$ schreibt man auch $q_i R q_{i+1}$ oder $q_i \rightarrow q_{i+1}$. Auf der Menge der Zustandsübergänge wird durch den Operationsbegriff eine Äquivalenzrelation bez. der Adressen definiert.

$$\sigma_{ij} = \{(d, d') \mid (a_i, d) R (a_j, d')\}$$

heißt Operation des Prozesses P. Ein Übergang (a_i, d) nach (a_j, d') ist mit σ_{ij} möglich, wenn $(d, d') \in \sigma_{ij}$ gilt.

In Zeichen: $(a_i, d) \xrightarrow{\sigma_{ij}} (a_j, d')$

Eine Operation $\sigma \in \Sigma_i = \{\sigma_{ij} \mid i \neq j\}$ heißt ausführbar im Zustand q, wenn $q = (a, d) \wedge a = a_i \wedge d \in \text{Dom}(\sigma)$.

In Zeichen: $\text{ausf}(\sigma, q)$

Die Betriebssystemmaschine

Die Betriebssystemmaschine ist definiert durch

$$\text{bsm} = (\mathbb{P}, D, \text{Var}, R, \text{prio}, \text{vtgl}, q_0)$$

- $\mathbb{P} = \{P_1, \dots, P_n\}$ ist eine Menge von sequentiellen Prozessen, deren Adreßmengen A_i paarweise disjunkt sind.
- D ist eine für alle Prozesse globale Datenmenge. Dieser ist eine globale Variablenmenge Var zugeordnet, deren Elemente $v \in \text{Var}$ Werte in einem Wertebereich $D_v \subset D$ annehmen können. Für $u \in \text{Var}$ wird mit $w(d, u)$ der Wert der Variablen u im Datenzustand $d \in D$ bezeichnet. Die den Datenbereichen D_i der sequentiellen Prozesse P_i zugeordneten Variablenmengen Var_i können in die globale Variablenmenge Var durch ihre Vereinigung abgebildet werden:

$$\text{Var} = \bigcup_{1 \leq i \leq n} \text{Var}_i$$

Es wird also von globalen Variablenbezeichnungen ausgegangen. Sollen verschiedene Bezeichner der einzelnen Variablenmengen die gleichen Variablen bezeichnen, so müssen diese mit einer Identifizierungsfunktion als identisch erklärt werden.

- Die Relation R beschreibt die möglichen Zustandsübergänge der Betriebssystemmaschine.

$$QE = \prod_{1 \leq i \leq n} A_i \times D \times H, \quad R \subset QE \times QE$$

Die Menge $H = H_{b1} \times H_{1a} \times H_{wa} \times H_{be}$, wobei die einzelnen H_i Teilmengen der Potenzmenge von $\mathbb{P}$ sind, dient zur Definition von Tätigkeitszuständen der Prozesse. Alle $h, h' \in H$ mit

$(a_1, \dots, a_n, d, h) R (a'_1, \dots, a'_n, d', h')$ sind Zerlegungen der Prozeßmenge $\mathbb{P}$.

- prio: Prioritätsrelation, vtgl: Verträglichkeitsrelation. Mit Σ_i : Menge aller Operationen des Prozesses P_i und $\Sigma = \bigcup_{1 \leq i \leq n} \Sigma_i$ gilt:

$$\text{prio} \subset \bar{P} \times \bar{P} \times \text{QE} \times \Sigma \times \Sigma$$

prio ist transitiv, antisymmetrisch und irreflexiv.

$$\text{vtgl} \subset \bar{P} \times \bar{P} \times \text{QE} \times \Sigma \times \Sigma$$

vtgl ist symmetrisch und reflexiv.

- $q_0 = (\bar{a}_0, D_0)$ ist die Menge der Anfangszustände. Dabei ist $D_0 \subset D$ die Anfangsdatenmenge mit folgenden Eigenschaften: Für jedes $d_0 \in D_0$ gibt es mindestens einen Prozeß P_i , dessen Anfangsadresse in der Projektion von $\bar{a}_0$ auf P_i enthalten ist und in dessen Anfangsdatenmenge D_{i_0} ein Element d_{i_0} enthalten ist, das in der Projektion von d_0 auf P_i sich ergibt.

Zur Definition der Semantik der Betriebssystemmaschine sollen zunächst die Tätigkeitszustände h_{bl} , h_{la} , h_{be} und h_{wa} erläutert werden. Ein Prozeß ist im Zustand blockiert, wenn keine seiner Operationen ausführbar ist. Dabei heißt ein Prozeß P_i im Zustand $q = (a_1, \dots, a_n, d)$ ausführbar, wenn für die Projektion $\Pi(q, i) = (a_i, d_i)$ gilt: Die Operation σ geht von a_i aus und $d_i \in \text{Dom}(\sigma)$.

$$P_i \in h_{bl}(q) \iff (\forall \sigma \in \Sigma) (\neg \text{ausf}(\sigma, q))$$

Ein Prozeß heißt bereit, wenn er, entsprechend der angegebenen Verträglichkeitsrelation, mit allen laufenden Prozessen verträglich ist.

$$P_i \in h_{bl} \wedge P_i \in h_{la} \wedge (\forall P_j, \sigma, \sigma') (i \neq j \wedge P_j \in h_{la} \wedge \sigma \in \text{bef}(P_i, q) \wedge \sigma' \in \text{bef}(P_j, q) \implies \text{vtgl}(P_i, P_j, q, \sigma, \sigma'))$$

($\text{bef}(P_j, q) = \Sigma_j =$ die Menge aller im Zustand $q = (a_1, \dots, a_n, d)$ von a_i ausgehenden Operationen des Prozesses P_j .)

Ist eine Prozeßoperation nicht mit der Operation eines laufenden Prozesses verträglich, so wird der Prozeß in den Zustand wartend versetzt. Dies geschieht z.B. dann, wenn aufgrund der Prioritätsrelation Operationen laufend werden, die mit den Operationen bereiter Prozesse nicht verträglich sind.

$$P \in h_{bl} \wedge P \in h_{la} \wedge P \in h_{be}$$

Der Zustand laufend der Betriebssystemmaschine wird durch die zwei Operationen START und END verändert. Die START-Operation ist durch folgende Zustandsänderung von q nach q' definiert:

$$q = (q, h_{bl}, h_{be}, h_{la}, h_{wa}), q' = (q', h'_{bl}, h'_{be}, h'_{la}, h'_{wa}), P \in \bar{P} \\ P \in h_{be} \wedge P \in h'_{la} \wedge (\forall P', \sigma, \sigma') (P' \in h_{be} \wedge \neg \text{prio}(P', P, q, \sigma, \sigma'))$$

Für die END-Operation gilt:

- $qe = (q, h_{bl}, h_{be}, h_{la}, h_{wa}), qe' = (q', h'_{bl}, h'_{be}, h'_{la}, h'_{wa}), P_i \in \bar{P}$
- $P_i \in h_{la} \wedge h'_{la} = h_{la} \setminus \{P_i\}$
- Falls für den Prozeß P_i gilt: $\Pi(q, i) = (a_i, d_i) \wedge \Pi(q', i) = (a'_i, d'_i)$, so existiert eine Operation σ zwischen den Adressen a_i, a'_i von P_i mit $d_i \sigma d'_i$
- Für jede Datenvariable aus $Var-Var_i$ gilt: $w(d, u) = w(d', u)$
- $h'_{la}, h'_{be}, h'_{bl}, h'_{wa}$ sind wie oben definiert, ebenso der Tätigkeitszustand von P_i .

Die Semantik der Betriebssystemmaschine wird über die Berechnungsfolgen

$\langle qe_i \mid 0 \leq i \leq m \rangle, m \in \mathbb{N}_0$
 definiert. $qe_i R qe_{i+1}$ ist Element einer Berechnung, wenn gilt:
 $qe_i \xrightarrow{START} qe_{i+1}$ oder $qe_i \xrightarrow{END} qe_{i+1}$
 und für qe_0 gilt das Initialisierungsaxiom, d.h., h_{wa} und h_{la} sind leer.

2.2 Informationsfluß

Zur Definition des Informationsflußbegriffs werden zwei Relationen der Betriebssystemmaschine, die semantische Änderung und die semantische Abfrage verwendet (Keramidis). Dabei sind die globalen Variablen zur Kommunikation zwischen Prozessen wichtig. Damit zwischen zwei Prozessen P_1 und P_2 Information fließen kann, muß P_1 zunächst eine globale Variable ändern und P_2 die Variable anschließend semantisch abfragen (bevor die Variable erneut geändert wird). Durch Verwendung mehrerer, nicht notwendig verschiedener Variablen kann der Informationsfluß auf eine abwechselnde Folge von semantischen Änderungen und Abfragen ausgedehnt werden. Die folgenden Abschnitte gehen darauf genauer ein.

Der definierte Informationsflußbegriff genügt nicht zur Beschreibung von Zugriffsschutzproblemen, da nicht die Art des

Informationsflusses, sondern nur Quelle (ändernder Prozeß) und Ziel oder Senke (abfragender Prozeß) verwendet werden. Deshalb wird der Begriff des bedingten Informationsflusses eingeführt, wo durch Prädikate zwischen verschiedenen Wegen des Informationsflusses unterschieden wird. Vom Informationsfluß wird der Begriff der Informationsübertragung unterschieden, wo nicht der Fluß zwischen Prozessen (allgemeiner: Operationenmengen) betrachtet wird, sondern der Fluß zwischen Variablen. Schließlich wird auch noch ein eigenes Prädikat zur Charakterisierung von Objektzugriffen eingeführt. Auch diese Aspekte werden im folgenden behandelt.

2.2.1 Semantische Abfrage bzw. Änderung

Zur Charakterisierung des Informationsflusses zwischen zwei Operationenmengen ist der lesende bzw. schreibende Zugriff auf gemeinsame Variablen notwendig. Deshalb wird zunächst definiert, wann eine Variable u semantisch geändert bzw. abgefragt werden kann. Das Attribut semantisch wird eingeführt, weil bei einer Sprache zur Beschreibung von Prozeßsystemen, deren Semantik mit Hilfe der Betriebssystemmaschine definiert wird, zwischen syntaktischer und semantischer Änderung bzw. Abfrage unterschieden werden muß. Z.B. gilt in der Anweisung

$x := \text{IF } a=0 \text{ THEN } b+c-c \text{ ELSE } x \text{ END};$

- x wird syntakt. abgefragt/geändert, wird semant. geändert
- a wird syntakt. abgefragt, wird semantisch abgefragt
- b wird syntakt. abgefragt, wird semantisch abgefragt
- c wird syntakt. abgefragt.

x wird nicht semantisch abgefragt, da beim Durchlaufen des ELSE-Zweigs der Wert der Variablen x keine Rolle spielt. Ob x semantisch geändert wird, hängt vom Datenzustand der Variablen a ab (ebenso die semantische Abfrage von b). c wird nicht semantisch abgefragt, da $b+c-c$ zu b vereinfacht werden kann (beim Ausschluß von Rundungsfehlern) und so beim Wert von x nicht in Erscheinung tritt. Die syntaktische Änderung/Abfrage ergibt sich aus dem syntaktischen Vorkommen einer Variablen auf der linken bzw. rechten Seite einer Anweisung. Die Unterscheidung

syntaktisch - semantisch ist interessant, da die semantische Änderung/Abfrage einer Variablen nicht immer syntaktisch in Erscheinung treten muß. Z.B. kann die Ausführungszeit eines Programms Informationen über Programmeigenschaften enthalten. Die Veränderung der Zeitvariablen ist aber nicht explizit im Programm sichtbar. Auf den Zusammenhang zwischen semantischer und syntaktischer Abfrage/Änderung wird in Kapitel 3 näher eingegangen, wo die Semantik einer Spezifikationssprache für asynchrone, geschützte abstrakte Datentypen mit der Betriebssystemmaschine definiert wird.

Die semantische Änderung einer Variablen ergibt sich aus der Änderung ihres Werts nach Ausführung einer Operation. (Die weiteren Betrachtungen beziehen sich auf eine gegebene Betriebssystemmaschine $bsm = (\bar{P}, D, Var, R, prio, vtgl, q_c)$.)

Definition (semantische Änderung): Eine Variable u kann von der Operation σ im Datenzustand d semantisch geändert werden, wenn gilt: $(\exists d' \in D)((d, d') \in \sigma \wedge w(d, u) \neq w(d', u))$

In Zeichen: $send(\sigma, d, u)$

Die semantische Abfrage einer Variablen u kann informell dadurch charakterisiert werden, daß sich Änderungen der Variablen u bei einer Operation σ in Änderungen der Variablen aus Var widerspiegeln. Um die Änderungen u eindeutig zuordnen zu können, darf nur die Variable u variiert werden. Die Eigenschaft kann durch folgendes einfache Bild veranschaulicht werden. Die betrachtete Operation wird als schwarzer Kasten aufgefaßt, der nur die Variable u als Eingabe hat und die Variablen $v_1, \dots, v_k \neq u$ als Ausgabe. Wird die Eingabevariable u variiert, so müssen sich zur semantischen Abfrage von u Änderungen in den Variablen $v_1, \dots, v_k$ ergeben.

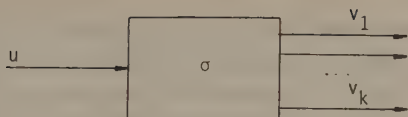


Abb. 32: Charakterisierung der semantischen Abfrage

Definition (semantische Abfrage): Eine Variable u kann in einem Datenzustand d von der Operation σ semantisch abgefragt werden, wenn gilt: $(\exists d_1 \in D) (EQ(d, d_1, u) \wedge (RS\ddot{A}(\sigma, d, u) \neq RS\ddot{A}(\sigma, d_1, u) \vee$

$$RSN\ddot{A}(\sigma, d_1, u) \neq RSN\ddot{A}(\sigma, d, u) [\overset{u}{w(d_1, u)}]))$$

In Zeichen: $abfr(\sigma, d, u)$

Dabei sind die Mengen $RS\ddot{A}$, $RSN\ddot{A}$, M und die Relation EQ wie folgt definiert:

$$EQ(d, d_1, u) \iff (\forall v \in Var) (v \neq u \implies w(d, v) = w(d_1, v))$$

$$RS\ddot{A}(\sigma, d, u) = \{d' \in D \mid d \sigma d' \wedge w(d, u) \neq w(d', u)\}$$

$$RSN\ddot{A}(\sigma, d, u) = \{d' \in D \mid d \sigma d' \wedge w(d, u) = w(d', u)\}$$

$$M[\overset{x}{y}] = \{d \mid w(d, x) = y \wedge (\exists d_1 \in M) (EQ(d, d_1, x))\}$$

Durch $EQ(d, d_1, u)$ wird die u -Äquivalenz zweier Datenzustände d_1 und d definiert. Sie gilt, wenn d und d_1 bis auf die Werte von u identisch sind.

$RS\ddot{A}(\sigma, d, u)$ und $RSN\ddot{A}(\sigma, d, u)$ bezeichnen die Mengen von Folgezuständen des Datenzustands d , in denen die Werte der Variablen u verschieden bzw. identisch sind.

Schließlich bezeichnet $M[\overset{x}{y}]$ die Menge, in denen alle Werte der Variablen x in den Zuständen der Menge M durch den Wert y ersetzt sind.

Dann kann eine Variable u im Datenzustand d von der Operation σ abgefragt werden, wenn bei Variation des Wertes von u

$$(\exists d_1 \in D) (EQ(d, d_1, u))$$

entweder sich die Werte der Variablen u selbst ändern

$$RS\ddot{A}(\sigma, d, u) \neq RS\ddot{A}(\sigma, d_1, u)$$

oder die Werte anderer Variablen als u sich ändern.

$$RSN\ddot{A}(\sigma, d_1, u) \neq RSN\ddot{A}(\sigma, d, u) [\overset{u}{w(d_1, u)}]$$

Das Attribut "kann" der semantischen Abfrage und Änderung ist

wichtig, da die Erfüllung der Prädikate $\text{abfr}(\sigma, d, u)$ oder $\text{änd}(\sigma, d, u)$ nur notwendige, nicht hinreichende Bedingung zur tatsächlichen Abfrage oder Änderung ist. Durch die Möglichkeit mehrerer Folgezustände zu einem Datenzustand d (relationale Operationen) muß nicht der im Prädikat betrachtete Folgezustand erreicht werden. Dies ist bereits ein Hinweis auf die Verwendung dynamischer Untersuchungen, um zu starke Einschränkungen des dynamischen Ablaufs wegen der bei statischen Untersuchungen notwendigen "Worst-case"-Annahmen zu vermeiden.

2.2.2 Direkter Informationsfluß

Im folgenden wird angenommen, daß Informationsflüsse nur über Variablen als Informationsträger erfolgen können. Informationen sind ohne Träger nicht denkbar, so daß der Zustand des Informationsträgers mit der Information selbst identifiziert werden kann. (Auch im vielzitierten Beispiel aus Sherlock Holmes, 'Der Hund von Baskerville', liegt im Nichtbellen eines Hundes nur deshalb Information, weil zwischen den Zuständen "Bellen" und "Nichtbellen" des Hundes als Informationsträger unterschieden werden kann.) Variablen bzw. ihre Zustände sind die Informationsträger in der Betriebssystemmaschine. Häufig ist der Informationsfluß zwischen verschiedenen Prozessen von Interesse. Das bedeutet, daß es gemeinsame Variablen dieser Prozesse geben muß, deren Zustände von Operationen der Prozesse geändert bzw. abgefragt werden können. Daraus ergibt sich wieder die Analogie zu Synchronisationsproblemen, da diese erst durch den möglichen konkurrierenden Zugriff mehrerer Prozesse auf gemeinsame Variablen entstehen.

Damit Information zwischen zwei Prozessen P_i und P_j über eine Variable $v \in \text{Var}$ fließt, muß der Wert der Variablen sowohl geändert als auch abgefragt werden. Wird eine Variable nur geändert, spricht man von einer Informationssenke, wird sie nur abgefragt, so heißt sie Informationsquelle. Die Änderung muß vor der Abfrage erfolgen, und zwischen der Änderung von P_i und der Abfrage von P_j darf v nicht von Prozessen P_k ($k \neq i$) geändert werden (siehe Bild 33).

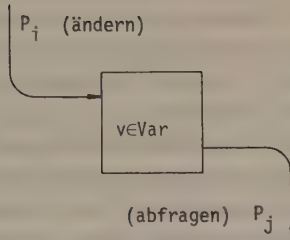


Abb. 33: Informationsfluß über eine Variable $v \in \text{Var}$

Mit der Definition der Relationen oaend und oabfr wird zunächst von einzelnen Datenzuständen bei der semantischen Abfrage/Änderung abstrahiert.

Definition (semantische Änderungsoperation): Eine Variable u kann von der Operation σ semantisch geändert werden, wenn gilt:

$$(\exists (d, d') \in \sigma) (w(d, u) \neq w(d', u))$$

In Zeichen: $\text{oaend}(\sigma, u)$

Definition (semantische Abfrageoperation): Eine Variable u kann von der Operation σ semantisch abgefragt werden, wenn gilt:

$$(\exists d, d_1 \in D) (EQ(d, d_1, u) \wedge \text{RSÄ}(\sigma, d, u) \neq \text{RSÄ}(\sigma, d_1, u) \vee \\ \text{RSNÄ}(\sigma, d_1, u) \neq \text{RSNÄ}(\sigma, d, u) [\overset{u}{w(d_1, u)}])$$

In Zeichen: $\text{oabfr}(\sigma, u)$

Mit folgenden Definitionen wird von konkreten Operationen abstrahiert und die semantische Änderung/Abfrage für Operationenmengen eingeführt.

Definition (semantische Änderungsmenge): Eine Variable u kann von der Operationenmenge Σ semantisch geändert werden, wenn gilt:

$$(\exists \sigma \in \Sigma) (\text{oaend}(\sigma, u))$$

In Zeichen: $\text{paend}(\Sigma, u)$

Definition (semantische Abfragemenge): Eine Variable u kann von der Operationenmenge Σ semantisch abgefragt werden, wenn gilt:

$$(\exists \sigma \in \Sigma) (\text{oabfr}(\sigma, u))$$

In Zeichen: $\text{pabfr}(\Sigma, u)$

Die Prädikate paend und pabfr sind unabhängig von Variablen-

zuständen und Operationen $\sigma \in \Sigma$. Wegen des relationalen Charakters der Operationen σ (ein Zustand d kann mehrere Nachfolgezustände d' mit $(d, d') \in \sigma$ haben), werden Zustandsübergänge in σ , die u ändern können, und Zustandsübergänge, die u nicht ändern können, gleich behandelt (analog abfragen). Die Verwendung des Existenzquantors bei der Definition der *oaend*- und *oabfr*-Prädikate liefert ebenfalls keine genauere Beschreibung der ändernden und abfragenden Operationen. Deshalb sollen die genannten Prädikate um eine Änderungs- bzw. Abfragebedingung erweitert werden, die die Zustandsübergänge, die zu einer Änderung bzw. Abfrage der Variablen u führen können, genauer klassifiziert. Die Bedingungen charakterisieren die Menge der Datenzustände, in denen semantische Änderungen/Abfragen möglich sind.

Definition (bedingte semantische Änderungsoperation): Eine Variable u kann unter der Bedingung B von der Operation $\sigma \in \Sigma$ geändert werden, wenn

$$(\exists (d, d') \in \sigma) (w(d, u) \neq w(d', u) \wedge B(d))$$

In Zeichen: $oaendb(\sigma, u, B)$

Definition (bedingte semantische Abfrageoperation): Eine Variable u kann unter der Bedingung B von der Operation $\sigma \in \Sigma$ abgefragt werden, wenn

$$(\exists d, d_1 \in D) (EQ(d, d_1, u) \wedge B(d) \wedge (RS\bar{A}(\sigma, d, u) \neq RS\bar{A}(\sigma, d_1, u) \vee RSN\bar{A}(\sigma, d_1, u) \neq RSN\bar{A}(\sigma, d, u) [w(d_1, u)^u]))$$

In Zeichen: $oabfrb(\sigma, u, B)$

Auch die Operationen einer Operationenmenge sollen durch Bedingungen genauer charakterisiert werden.

Definition (bedingte semantische Änderungsmenge): Eine Variable u kann unter der Bedingung B von der Operationenmenge Σ geändert werden, wenn

$$(\exists \sigma \in \Sigma) (oaendb(\sigma, u, B))$$

In Zeichen: $paendb(\Sigma, u, B)$

Definition (bedingte semantische Abfragemenge): Eine Variable u kann unter der Bedingung B von der Operationenmenge Σ abgefragt werden, wenn

$$(\exists \sigma \in \Sigma) (oabfrb(\sigma, u, B))$$

In Zeichen: $pabfrb(\Sigma, u, B)$

Die bedingten Prädikate x_b ($x \in \{oaend, oabfr, paend, pabfr\}$) sind Verallgemeinerungen der Prädikate x . Es gilt nämlich $x_b = x$, falls die Bedingung B den Wahrheitswert TRUE hat. Deshalb sind im folgenden mit Änderungs- und Abfrageprädikaten nur die bedingten Prädikate gemeint.

Die bisher eingeführten Prädikate bezeichnen bereits spezielle Informationsflüsse. Diese sind charakterisiert durch den Informationsfluß zu einer Variablen u bei den Änderungsprädikaten und von einer Variablen u bei den Abfrageprädikaten. In den Änderungs- bzw. Abfragemengen kann durch Spezialisierung auf die Operationenmengen eines Prozesses P zusätzlich die Informationsquelle bzw. -senke spezifiziert werden, nämlich der angegebene Prozeß P . Wie bereits bei den ändernden und abfragenden Operationen können die möglichen Datenzustände durch eine Bedingung $B_1 \wedge B_2$ (s.u.) charakterisiert werden. Mit Hilfe der definierten Prädikate wird im folgenden der direkte Informationsfluß über eine Variable u definiert.

Definition (direkter Informationsfluß): Unter den Bedingungen B_1 und B_2 kann zwischen den Operationenmengen Σ_1 und Σ_2 über die Variable u Information direkt fließen, wenn

$$paendb(\Sigma_1, u, B_1) \wedge pabfrb(\Sigma_2, u, B_2)$$

In Zeichen: $\inf(\Sigma_1, \Sigma_2, u, B_1, B_2)$

Durch Spezialisierung der Operationenmengen sind spezielle Informationsflüsse beschreibbar. Mit Σ_{P_i} wird im weiteren die Menge aller Operationen des Prozesses P_i bezeichnet.

- $\Sigma_1 = \Sigma_2 = \bigcup_{1 \leq i \leq n} \Sigma_{P_i}$: Damit werden alle Informationsflüsse über die Variable u einer gegebenen Betriebssystemmaschine charakterisiert.
- $\Sigma_1 = \{\sigma_1\}, \Sigma_2 = \{\sigma_2\}$: Damit wird der direkte Informationsfluß über zwei Operationen σ_1 und σ_2 , die auf der Variablen u operieren, gekennzeichnet.
- $\Sigma_1 = \Sigma_{P_k}, \Sigma_2 = \Sigma_{P_j}, P_k, P_j \in P$: Damit ist der Informationsfluß von Prozeß P_k zu Prozeß P_j über die Variable u einer gegebenen Betriebssystemmaschine gekennzeichnet.

Die Annahme, daß zum Informationsfluß zwischen verschiedenen Prozessen eine gemeinsame Variable notwendig ist, wird durch

die in den Prädikaten paendb und pabfrb vorkommende gleiche Variable u ausgedrückt. Die zweite Annahme, daß zum Informationsfluß sowohl eine Änderungs- als auch eine Abfrageoperation gehören, wird ebenfalls durch die zwei Bedingungen des inf -Prädikats ausgedrückt. Da keine Annahmen über die Operationenmengen gemacht werden, können auch Informationsflüsse innerhalb eines Prozesses ($\Sigma_1 = \Sigma_2 = \Sigma_{P_k}, P_k \in \bar{P}$) beschrieben werden. Die Informationsflußdefinition geht davon aus, daß zunächst die ändernde und dann die abfragende Operation ausgeführt wird und daß keine weiteren Änderungen dazwischen vorkommen. Für eine Darstellung der genannten Bedingungen müßte das inf -Prädikat nicht für Operationenmengen, sondern für Berechnungsfolgen einer Betriebssystemmaschine definiert werden. Dieser Detaillierungsgrad wird aber im weiteren nicht benötigt und deshalb nicht eingeführt.

Variablenzugriff

Mit dem Prädikat $\text{inf}(\Sigma_1, \Sigma_2, u, \text{TRUE}, \text{TRUE})$ kann ausgedrückt werden, ob die Variable u als Informationsträger verwendet wird. u wird dann nicht dazu verwendet, wenn die Variable weder geändert noch abgefragt wird. Auch bei nur abfragenden oder nur ändernden Operationen dient u nur als Informationsquelle oder -senke, nie als Informationsträger (siehe Informationsquelle/-senke am Anfang dieses Abschnitts). Um die Zugriffe aus einer Operationenmenge auf eine Variable u zu charakterisieren, genügt es, die Prädikate paendb und/oder pabfrb zu verwenden, da mit der Gültigkeit des inf -Prädikats sowohl die semantische Änderung als auch die semantische Abfrage impliziert wird. Durch Reduktion der Zugriffsarten auf die elementaren Operationen "ändern" und "abfragen" lassen sich beliebige Zugriffsarten beschreiben, da diese ebenfalls auf die elementaren Funktionen "lesen" und "schreiben" zurückgeführt werden müssen (elementare Speicherzugriffe).

Definition (Variablenzugriff): Mit der Operationenmenge Σ ist ein Zugriff auf die Variable u unter der Bedingung B möglich, wenn

$$\text{paendb}(\Sigma, u, B) \vee \text{pabfrb}(\Sigma, u, B)$$

In Zeichen: $\text{acc}(\Sigma, u, B)$

Der wesentliche Unterschied zum Informationsfluß liegt also darin, daß für den Zugriff nicht zwischen den Zugriffsarten "ändernd" oder "abfragend" unterschieden wird. Der Variablenzugriff ist aber notwendige Voraussetzung für den direkten Informationsfluß. Bei genauerer Untersuchung stellt sich heraus, daß das Zugriffsprädikat $\text{acc}(\Sigma, u, B)$ unter der Bedingung

$\text{paendb}(\Sigma, u, B) \text{ EXOR } \text{pabfrb}(\Sigma, u, B)$

nur Informationssenken ($\text{paendb}(\Sigma, u, B) = \text{TRUE}$) oder nur Informationsquellen ($\text{pabfrb}(\Sigma, u, B) = \text{TRUE}$) beschreibt. Diese sind bei der Betrachtung zyklischer Prozeßsysteme ohne Bedeutung (siehe Keramidis).

Satz (Äquivalenz von inf- und acc-Prädikat): In einem Prozeßsystem ohne Informationsquellen und -senken sind inf- und acc-Prädikat derselben Operationenmenge und unter derselben Bedingung B äquivalent.

Beweis: " $\rightarrow$ " $\text{inf}(\Sigma, \Sigma, u, B, B) \rightarrow \text{paendb}(\Sigma, u, B) \wedge \text{pabfrb}(\Sigma, u, B)$
 $\rightarrow \text{paendb}(\Sigma, u, B) \vee \text{pabfrb}(\Sigma, u, B)$
 $\rightarrow \text{acc}(\Sigma, u, B)$

" $\leftarrow$ " $\text{acc}(\Sigma, u, B) \rightarrow \text{paendb}(\Sigma, u, B) \vee \text{pabfrb}(\Sigma, u, B)$
 wegen $(\neg \exists u \in \text{Var}) (\text{paendb}(\Sigma, u, B) \text{ EXOR } \text{pabfrb}(\Sigma, u, B))$
 (Voraussetzung ohne Quellen und Senken)
 $\rightarrow \text{paendb}(\Sigma, u, B) \wedge \text{pabfrb}(\Sigma, u, B)$
 $\rightarrow \text{inf}(\Sigma, \Sigma, u, B, B)$

Berücksichtigung von Informationsquellen und -senken

Informationsquellen bzw. -senken sind zur Modellierung der Schnittstellen eines Prozeßsystems zu den umgebenden Komponenten, insbesondere der E/A-Schnittstellen, geeignet. Informationsquellen sind z.B. Eingabevariablen, die die von außen kommenden Werte enthalten, während Senken die Ausgabeparameter repräsentieren, die z.B. über ein druckendes Gerät ausgegeben werden.

Das System ADEPT-50 (siehe 1.2.1) zeigt deutlich, daß bei der Untersuchung von Informationsflüssen Ein-/Ausgabeparameter bzw. Initialisierungswerte einbezogen werden müssen. Dazu wird dort der Objekttyp "Terminal" in das Modell aufgenommen, der über den Informationsfluß vom und zum Terminal entscheidet.

Obiger Äquivalenzsatz gilt nur, wenn das Prozeßsystem keine Informationsquellen oder -senken enthält. Da diese aber i.allg. berücksichtigt werden müssen, soll der Satz allgemeiner formuliert werden. Dazu wird zunächst die Variablenmenge Var in zwei Mengen V_{inf} und V_{acc} aufgeteilt, so daß gilt: $V_{inf} \cup V_{acc} = Var$.

$$V_{inf} = \{v \in Var \mid inf(\Sigma, \Sigma, v, TRUE, TRUE)\}$$

$$V_{acc} = \{v \in Var \mid acc(\Sigma, v, TRUE)\}$$

Mit der Voraussetzung $V_{inf} \cap V_{acc} = \emptyset$ gilt obiger Äquivalenzsatz für alle Variablen aus V_{inf} . Informationsquellen und -senken sind in V_{acc} enthalten und können damit im Betriebssystemmaschinensmodell berücksichtigt werden. Der Äquivalenzsatz lautet dann wie folgt:

Satz: In einem Prozeßsystem sind für alle Variablen u aus V_{inf} und für die gleiche Operationenmenge inf- und acc-Prädikat äquivalent, wenn $V_{inf} \cap V_{acc} = \emptyset$.

2.2.3 Informationsübertragung und allgemeiner Informationsfluß

Bei der Definition des direkten Informationsflusses wird der Aspekt des Informationsträgers durch eine Variable hervorgehoben. Ein anderer Aspekt ist die Informationsübertragung zwischen zwei Variablen mit einer geeigneten Operationenmenge. Dieser Aspekt wird in folgender Definition berücksichtigt.

Definition (direkte Informationsübertragung): Von der Variablen u zur Variablen v wird, unter den Bedingungen B_1 und B_2 durch die Operationenmenge Σ_{P_i} eines Prozesses $P_i \in \bar{P}$ Information direkt übertragen, wenn

$$pabfrb(\Sigma_{P_i}, u, B_1) \wedge paendb(\Sigma_{P_i}, v, B_2)$$

In Zeichen: $trans(u, v, \Sigma_{P_i}, B_1, B_2)$

Wichtig an der Definition ist, daß Informationsübertragung nur für die Operationenmenge eines Prozesses definiert ist. Die Abfrage der Variablen u durch einen Prozeß P_i und die Änderung der Variablen v durch einen Prozeß P_j ($i \neq j, \forall u$) führt nur dann zur Informationsübertragung, wenn dazwischen ein Informationsfluß über eine gemeinsame Variable w zwischen Σ_{P_i} und Σ_{P_j} stattfindet.

Definition (allgemeiner Informationsfluß zwischen zwei Prozessen): Zwischen zwei Prozessen P_i und P_j aus $\bar{P}$ kann allgemein Information fließen, wenn gilt:

$$(\exists \Sigma_1, \dots, \Sigma_n)(\exists u_1, \dots, u_{n-1})(\Sigma_1 = \Sigma_{P_i} \wedge \Sigma_2 = \Sigma_{P_j} \wedge n \geq 2 \wedge \\ (\forall k \in \{1, \dots, n-1\})(\inf(\Sigma_k, \Sigma_{k+1}, u_k, \text{TRUE}, \text{TRUE})))$$

In Zeichen: $\text{infa}(P_i, P_j)$

Allgemeiner Informationsfluß zwischen zwei Prozessen ist also dann möglich, wenn eine Folge von ändernden und abfragenden Operationen existiert, in der, von P_i ausgehend, die (nicht notwendig verschiedenen) Variablen $u_1, \dots, u_{n-1}$ abwechselnd geändert und abgefragt werden, wobei die Folge mit einer Abfrageoperation von P_j endet. Die Folge wird dadurch verknüpft, daß jede Operationenmenge Σ_i ($i \in \{2, \dots, n-1\}$) u sowohl abfragen als auch ändern kann. Als Spezialfall ergibt sich für $n=2$ der direkte Informationsfluß zwischen den Prozessen P_i und P_j über die Variable u_1 .

Erweiterte Betriebssystemmaschine

Da im folgenden das inf -Prädikat einer Betriebssystemmaschine verwendet wird, soll das Modell erweitert werden.

$$\text{bsm} = (\bar{P}, D, \text{Var}, R, \text{prio}, \text{vtgl}, \text{inf}, q_0)$$

heißt erweiterte Betriebssystemmaschine. Das inf -Prädikat beschreibt die in der erweiterten Betriebssystemmaschine möglichen Informationsflüsse. Da im weiteren nur noch die erweiterte Betriebssystemmaschine verwendet wird, wird nur von der Betriebssystemmaschine gesprochen.

2.3 Vergleich mit bekannten Modellen

Das Verbandsmodell von Denning

Wie in Abschnitt 1.1.3 beschrieben, besteht das Schutzmodell GS3 aus Subjekten, Objekten, Sicherheitsklassen, dem lub-Operator und der Informationsflußrelation: $GS3 = (S, O, SC, \oplus, ->)$. Folgende Betriebssystemmaschine modelliert GS3. Die Menge der Subjekte kann mit den Prozessen $P_i \in \mathbb{P}$ identifiziert werden. Da die Menge der Objekte nicht einzelnen Subjekten zugeordnet ist, wird die Menge Var der Betriebssystemmaschine als Objektmenge aufgefaßt und ist damit für alle Prozesse gleich (globale Variablenmenge). Der Datenbereich D der Variablen ist mit der Menge der Sicherheitsklassen SC zu identifizieren, die Abbildung w, die jeder Variablen einen Wert zuordnet, ist also durch $w: O \rightarrow SC$ gegeben. Daraus wird deutlich, daß sich GS3 auf Informationsflußuntersuchungen beschränkt. Das Modell ist unabhängig von Werten der Variablen, wenn man die Sicherheitsklassen entfernt. Die Sicherheitsklassen sind Repräsentanten für Teilmengen $D' \subset D$ der Datenbereiche der Variablen aus Var. GS3 betrachtet nur die eingeschränkten Datenbereiche D' . Auf D' ist eine Verbandsstruktur definiert. Die Operationen σ_{ij} eines Prozesses sind als Kompositionen des lub-Operators auf D' gegeben. Zwischen dem trans-Prädikat der Betriebssystemmaschine und der Informationsflußrelation " $\rightarrow$ " in GS3 besteht folgender Zusammenhang:

$$\text{trans}(u, v, \perp, \text{TRUE}, \text{TRUE}) < \rightarrow \underline{u} \rightarrow \underline{v}$$

Die vorgestellte Beschreibung des Verbandsmodells und der Informationsübertragung mit dem trans-Prädikat der Betriebssystemmaschine lassen folgende Nachteile von GS3 erkennen:

- Die Zuordnung zwischen Operationen und Wertänderungen der Variablen aus Var ist nicht im Modell enthalten.
- Der vom Variablenwert abhängige Informationsfluß muß getrennt spezifiziert werden, da die Wertänderungen nicht im Modell enthalten sind, sondern nur die den Änderungen zugeordneten Verbandsoperationen (lub-Operator).
- Durch die Eigenschaft SCO existiert in GS3 keine eindeutige Trennung zwischen aktiven und passiven Einheiten, so daß die

Begriffe Informationsfluß und Informationsübertragung vermischt werden.

- Es wird nur der Aspekt der Informationsübertragung berücksichtigt.
- Es werden nur solche Informationsflüsse dargestellt, die sich durch Verbände beschreiben lassen.

Das Modell der Schutzmatrix

Beim Modell der Schutzmatrix werden keine Informationsflüsse untersucht, sondern die Aufrufe von Operationen auf den gegebenen Objekten. Zusätzlich wird in der Schutzmatrix eine Beziehung zwischen den aufrufbaren Operationen und den Subjekten hergestellt. Zur Beschreibung der Schutzmatrix in der Betriebssystemmaschine genügt es, das acc-Prädikat mit einer eingeschränkten Operationenmenge zu verwenden. Jeder Operation der Operationenmenge muß ein Recht r eindeutig zugewiesen werden. Der Datenzustand der Variablen aus Var besteht aus Paaren (i, r) , wobei i einen Prozeßindex und r ein Recht bezeichnet. Dann gilt der Zusammenhang:

$$\text{acc}(\Sigma_{P_i}, u, B(P_i, r, w(d, u))) \longleftrightarrow r \in R' \wedge (P_i, u, R') \in SM$$

$$\text{wobei: } B(P_i, r, w(d, u)) \longleftrightarrow (i, r) \in w(d, u) \wedge$$

$$(\forall \sigma \in \Sigma_{P_i}) (\exists r \in R) (\exists f: \Sigma_{P_i} \rightarrow R) f \text{ bijektiv}$$

Es zeigt sich, daß in der Schutzmatrix die Bedingung B über den Zugriff auf die Variable u entscheidet. Die Operationenmenge bleibt bis auf die ihr zugeordnete Rechtemenge unberücksichtigt. Die Äquivalenz gilt nur, wenn allen Operationen ein Recht, das in der Bedingung B untersucht wird, zugeordnet ist. Ansonsten erlaubt die Betriebssystemmaschine mehr Zugriffe als die Schutzmatrix. Daran wird deutlich, daß das Modell der Schutzmatrix nur für solche Operationenmengen anwendbar ist, wo keine "verborgenen" Änderungs- oder Abfrageoperationen existieren. "Verborgene" ist dabei durch die eindeutige Zuordnung von Rechten zu Operationen definiert.

Der Sicherheitsbegriff von Popek

Die vorgestellten Prädikate $paendb$ und $pabfrb$ ähneln den Prädikaten "tatsächliches Lesen/Schreiben", wie sie Popek in /POPE78/ zur Definition der Sicherheit eines Betriebssystemkerns angegeben hat (siehe 1.3.2). Obwohl die dort angegebenen Prädikate einen allgemeinen Charakter haben, sind sie durch das umgebende Prozeßmodell, das nur sequentielle Prozesse ohne Interaktionen beschreibt, stark eingengt. Die Operationen sind funktional und vereinfachen dadurch die definierten Prädikate. Bezüge zur Synchronisation, die die Betriebssystemmaschine durch Berücksichtigung globaler Variablen enthält, sind nicht formulierbar.

Eigenschaften des inf-Prädikats

Die genannten drei Beispiele zeigen die wesentlichen Eigenschaften des inf-Prädikats der Betriebssystemmaschine.

- Der Informationsfluß (die Informationsübertragung, Objektzugriffe) wird in ein allgemeines Modell zur Beschreibung asynchroner Systeme eingebettet und erlaubt damit die Beschreibung asynchroner, geschützter Systeme.
- Die Prädikate sind unabhängig von Implementationsaspekten, da sie unabhängig von Interpretationen der Datenbereiche definiert sind.
- Die Allgemeinheit der Betriebssystemmaschine impliziert auch die Allgemeinheit der erweiterten Betriebssystemmaschine.
- Die Beschreibung der bisher bekanntesten Schutzmodelle (Verbandsmodell, Schutzmatrix) mit Hilfsmitteln der erweiterten Betriebssystemmaschine soll informell die Anwendbarkeit der Prädikate auf beliebige Schutzprobleme begründen. Ein formaler Beweis der Allgemeinheit der Prädikate kann nicht erbracht werden, da der Begriff des Schutzproblems erst im jeweiligen Modell definiert wird.

3 Schutzkonstrukte einer operationellen Spezifikationssprache

Die Verwendung von Software-Engineering-Methoden bei der Entwicklung von Software-Systemen ist Stand der Technik. Viele der verwendeten Methoden basieren auf einer Top-down-Strategie. Die Probleme werden zunächst analysiert und spezifiziert. Aus der Analyse ergibt sich eine Systemstruktur, die sich sowohl in der Spezifikation als auch in der daraus entstehenden Implementation widerspiegelt. Manche Methoden enthalten Verifikations-schritte, um z.B. die Implementation bez. der Spezifikation zu verifizieren. In den meisten Fällen wird lediglich validiert. Zu den Validationsmöglichkeiten gehören das Prüfllesen von Spezifikationen und Programmen und das Testen.

In Kapitel 1.3.1 wurden zwei formale Methoden, HDM und FDM, zur Spezifikation, Implementation und Verifikation von Software-Systemen vorgestellt. Beide Methoden werden durch eine Anzahl halbautomatischer (oft auch interaktiv genannter) Werkzeuge unterstützt. Dazu gehören Spezifikationssprachenprozessoren, Generatoren für Verifikationsbedingungen und Beweiser. Keine der dabei verwendeten Spezifikationssprachen enthält Konstrukte zur Formulierung von Schutzproblemen. Das eigentlich zu spezifizierende Software-System wird von den immanenten Schutzproblemen getrennt oder an die Schutzstrategie angepaßt. Als Beispiel kann der in /CHEH81/ beschriebene "Niedrigwasser"-Mechanismus genannt werden. HDM wird als einzige der dort beschriebenen Methoden von einem Werkzeug unterstützt, das eine gegebene Spezifikation auf Informationsflüsse untersucht. Die Untersuchungen sind auf das Verbandsmodell beschränkt.

Die Bedeutung von Schutzproblemen gerade bei der Software-Entwicklung wurde bereits hervorgehoben. Einerseits beeinflussen Schutzaspekte viele Entwurfsentscheidungen eines Software-Systems. Deshalb werden z.B. Betriebssysteme um einen Sicherheitskern herum aufgebaut. Auch das nachträgliche Erweitern eines Systems um Schutzmaßnahmen ist kaum möglich. Andererseits sind Schutzaspekte stark problemabhängig, so daß sie explizit bei der Problemspezifikation sichtbar sein sollten, um ihre Korrektheit überprüfen zu können. Deshalb sollte eine Methode

zur Spezifikation von Software-Systemen auch Möglichkeiten zur expliziten Spezifikation von Schutzproblemen enthalten. Die Methode sollte auch die korrekte Implementation von Software-Systemen unterstützen. Zum formalen Nachweis der Korrektheit, die in zuverlässigen Systemen vorausgesetzt wird (siehe Kapitel 1.1), ist auch die formale Spezifikation notwendig.

Deshalb wird in den weiteren Kapiteln auf die bei Keramidis vorgestellte Methode zur Spezifikation und korrekten Implementation asynchroner verteilter Prozeßsysteme Bezug genommen. Die Methode baut auf der in Kapitel 2 eingeführten Betriebssystemmaschine auf. Das vorliegende Kapitel 3 erweitert eine operationelle Spezifikationssprache zum Spezifizieren asynchroner abstrakter Datentypen um Konstrukte zur Formulierung von Schutzproblemen, so daß asynchrone, geschützte abstrakte Datentypen spezifizierbar werden. Die Sprache wird im weiteren mit SPAGAT (Spezifikationssprache für asynchrone, geschützte abstrakte Datentypen) bezeichnet. Der erste Abschnitt beschreibt die bei Keramidis definierte Spezifikationssprache, die im zweiten Abschnitt durch Schutzkonstrukte zu SPAGAT erweitert wird. Die Konstrukte basieren auf den im zweiten Kapitel eingeführten inf- und acc-Prädikaten. Der dritte Abschnitt soll anhand von Beispielen die Anwendungsmöglichkeiten der Schutzkonstrukte aufzeigen. Schließlich werden im vierten Abschnitt noch spezielle Aspekte bei der Spezifikation von Schutzproblemen untersucht: Das Vollständigkeitspostulat und versteckte Kanäle.

3.1 Spezifikationssprache einer operationellen Spezifikations- methode

Aufbauend auf der Betriebssystemmaschine hat Keramidis in /KERA82/ eine Methode zur Spezifikation und korrekten Implementation abstrakter Datentypen vorgestellt. Ein abstrakter Datentyp wird dabei als Datenstruktur mit darauf definierten Operationen aufgefaßt. Die Methode unterstützt die Datenabstraktion, da die Datenstruktur unabhängig von konkreten Implementationsstrukturen beschreibbar ist. Die Methode unterstützt die Funktionsabstraktion, da die Operationen eines abstrakten Typs mit Hilfe von Vor- und Nachbedingungen unabhängig von Implementationsalgorithmen definiert werden. Zur Implementation eines abstrakten Typs sind beliebige Programmiersprachen denkbar. Unterschiede ergeben sich aber bei der Verifikation der Programme und bei der Programmerstellung selbst. Programmiersprachen, die ein Modulkonzept enthalten und die strukturierte Programmierung unterstützen, sind besonders geeignet. Einerseits können mit dem Modulkonzept abstrakte Datentypen einfach implementiert und, falls das Modulkonzept das Prinzip des "information hiding" berücksichtigt, voneinander isoliert werden. Andererseits gibt es für PASCAL-artige Programmiersprachen bereits Korrektheitskalküle, basierend auf Prädikamentransformationen, die über die Prädikate einfach mit der Spezifikationsmethode verknüpft werden können. Ein Algorithmus ist dann die korrekte Implementation einer spezifizierten Operation, wenn seine Vor- und Nachbedingungen in vorgegebenen Beziehungen zu den Vor- und Nachbedingungen der Spezifikation stehen.

Für die grob skizzierte Spezifikationsmethode gibt es eine Spezifikationssprache, die im vorliegenden Abschnitt beschrieben wird, so daß bei Einführung der Schutzkonstrukte in zweiten Abschnitt darauf Bezug genommen werden kann. Außerdem wird informell ein Bezug zur Betriebssystemmaschine hergestellt.

3.1.1 Die Spezifikationssprache

Durch 'ABSTRACT TYPE SPECIFICATION

module_name (parameter_list)'

wird die Spezifikation eines abstrakten Typs eingeleitet (Abb. 34 zeigt ein Beispiel). 'module_name' ist der Bezeichner des Datentyps. Durch die in Klammern stehenden Parameter ist der abstrakte Typ parametrisierbar (generisch). Durch die Typparameter wird also nicht eine konkrete Datenstruktur, sondern ein Satz von Datenstrukturen einem abstrakten Typ zugeordnet. Die aktuellen Parameter, die bei der Erzeugung von Objekten des abstrakten Typs angegeben werden, müssen den möglichen Parameterwerten entsprechen. In einem 'IMPORT'-Teil werden alle aus anderen Datentypen benötigten Operationen angegeben. Eine 'EXPORT'-Klausel analog zu Modula-2 gibt es nicht, da die möglichen Exporte sich auf den Typnamen und die definierten Operationen beschränken. Der Teil 'ABSTRACT REPRESENTATION' dient zur Definition einer (eventuell von Typparametern abhängigen) Datenstruktur des abstrakten Typs, mit deren Hilfe die Vor- und Nachbedingungen der Operationen festgelegt werden. Die Datenstruktur ist unabhängig von Implementationsdetails und soll die problemnahe Beschreibung der Operationseigenschaften unterstützen. Ein Objekt mit dieser Struktur soll im weiteren auch abstraktes Objekt heißen. Enthält die Repräsentation mehrere Teilobjekte, so sind diese innerhalb der Spezifikation einzeln ansprechbar. Sie können als Komponenten des abstrakten Objekts aufgefaßt werden und sollen deshalb als abstrakte Komponenten bezeichnet werden. In den Teilen 'TYPES' und 'PARAMETERS' sind lokale Typen und formale Parameter der Typoperationen spezifizierbar, die auch in den Synchronisationsbedingungen vorkommen können. Im Teil 'DEFINITIONS' sind Abkürzungen für wiederholt verwendete oder längere Bedingungsteile definierbar. Während innerhalb der Spezifikation Aussagen über die Komponenten der abstrakten Repräsentation gemacht werden, wird der Typname zur Deklaration von Objekten des abstrakten Typs verwendet. Z.B. sei das Objekt 'buffer' deklariert mit:

buffer: nezev(20, INTEGER);

Der Inhalt des abstrakten Objekts ist nur über die Operationen des zugehörigen Typs erreichbar. Dazu wird der Objektname mit

der aufzurufenden Operation konkateniert, z.B.:

```
buffer.insert(elem);
```

Obwohl bei der Spezifikation des abstrakten Typs nur Aussagen über die Struktur der abstrakten Repräsentation gemacht werden, wird von einem abstrakten Objekt gesprochen, da die Struktur eine Schablone für jeweils ein Objekt des Typs darstellt. "Abstraktes Objekt" bezeichnet also immer ein Objekt des abstrakten Typs oder die im Typ verwendete Struktur.

Schließlich enthält der Teil 'OPERATIONS' die Operationen eines abstrakten Typs. Nach dem Operationsnamen stehen in Klammern die Eingabeparameter und, durch "->" getrennt, die Ausgabeparameter. Die Konstrukte 'PRE' und 'NBL' einer Operation enthalten Bedingungen, unter denen die Operation ausführbar sein soll. Aber während eine nichterfüllte 'PRE'-Bedingung zum Operationsabbruch führt, wird bei einer nichterfüllten 'NBL'-Bedingung die Operation lediglich blockiert. Sobald die Bedingung aufgrund von Operationsaufrufen anderer Prozesse erfüllt ist, erfolgt die Deblockierung. Die Auswirkung einer Operation beschreibt jeweils der 'EFFECTS'-Teil, in dem die Beziehungen, die nach Ausführung der Operation zwischen den Ein-/Ausgabeparametern, dem Zustand des abstrakten Objekts vor Operationsausführung und dem Zustand nach Operationsausführung gelten, prädikatenlogisch definiert sind. Mehrere 'EFFECTS'-Bedingungen können zur Übersichtlichkeit durch "---" getrennt werden. "---" entspricht der Und-Verknüpfung der Bedingungen (analog wird "---" in den 'PRE'- und 'NBL'-Teilen gebraucht). Zur Unterscheidung des alten und des neuen Zustands einer abstrakten Komponente wird der Komponentenname zur Bezeichnung des neuen Zustands mit einem Apostroph versehen.

Zur Beschreibung des 'SYN'-Teils, der die Abarbeitungsstrategie der Operationen eines Datentyps, die auf einem Objekt des Typs operieren, bestimmt, muß der Aktivitätsbegriff eingeführt werden. Eine Aktivität beginnt mit dem Aufruf einer Operation eines abstrakten Typs und endet mit der Rückkehr aus der Operation. Im 'SYN'-Teil werden Synchronisationsbedingungen für die Aktivitäten eines abstrakten Typs definiert. Dabei werden nur Aktivitäten betrachtet, die dasselbe Objekt bearbeiten. Bezie-

hungen zwischen abstrakten Objekten desselben Typs sind nicht vorgesehen. Die Bedingungen sind abhängig von den Operationsparametern, dem Objektzustand und von weiteren Aktivitäten am Objekt. Aktivitäten werden durch den Operationsnamen bezeichnet. Da Aktivitäten nicht im 'OPERATIONS'-Teil vorkommen und Operationen nicht außerhalb davon, entstehen keine Konflikte zwischen einem Aktivitäts- und einem Operationsbezeichner. Sind die Synchronisationsbedingungen von Parametern abhängig, so müssen diese ebenfalls in den Aktivitätsnamen übernommen werden. Der 'SYN'-Teil kann die Konstrukte 'COMP' ("compatibility") und 'PRIO' ("priority") enthalten. Das 'COMP'-Konstrukt legt fest, welche Aktivitäten miteinander verträglich sind, also gleichzeitig an einem gemeinsamen Objekt aktiv sein können. Das 'PRIO'-Konstrukt legt die Reihenfolge fest, in der laufbereite Aktivitäten (nicht blockiert und mit allen laufenden verträglich) laufend werden können. Die Begriffe "laufend", "bereit", "wartend" und "blockiert" entsprechen den in Kapitel 2 vorgestellten Prozeßzuständen. Die Aktivitäten werden syntaktisch über Aktivitätsparameter (Parameter vom Typ 'ACTIVITY') angesprochen, die als formale Parameter der Konstrukte 'COMP' und 'PRIO' spezifiziert werden. Der Eindeutigkeit wegen sind Operationsparameter in Synchronisationsbedingungen mit dem zugehörigen Aktivitätsparameter als Präfix zu versehen (z.B. 'al.pr').

Das in Bild 34 angegebene Beispiel eines gemeinsamen Puffers, bei dem gleichzeitig zwei verschiedene Elemente eingefügt und entfernt werden können, soll die vorgestellten Konstrukte verdeutlichen. Eine mögliche Anwendung des Typs 'nezev' (N-Erzeuger-Ein-Verbraucher) ist ein Spool-Puffer, der die Druckaufträge an ein Druckgerät serialisiert (Bild 35). Der abstrakte Datentyp 'nezev' ist parametrisiert bez. der Anzahl der Pufferelemente ('length') und ihres Typs ('t'). Als problemnahe Beschreibung ('ABSTRACT REPRESENTATION') wird der Puffer als Sequenz von 't'-Elementen dargestellt, die anfänglich leer ist. In 'PARAMETERS' werden die formalen Parameter der Operationen (und damit auch der Aktivitäten) spezifiziert. Das 'COMP'-Konstrukt besagt, daß Einfügen und Entfernen eines Pufferelements parallel ablaufen können, wenn der Puffer mehr als ein

```

ABSTRACT TYPE SPECIFICATION nezev (length: CARDINAL; t: TYPE);

  ABSTRACT REPRESENTATION
    nezev : SEQUENCE OF t INIT EMPTY_SEQUENCE;

  PARAMETERS
    pr: CARDINAL;
    elem: t;
    a1,a2: ACTIVITY;

  SYN
    COMP(a1,a2) <=> a1∈insert ∧ a2∈remove ∧ LENGTH(nezev)>1;
    PRIO(a1,a2) <=> a1∈insert(pr) ∧ a2∈insert(pr) ∧ a1.pr>a2.pr;

  OPERATIONS
    insert(pr,elem);
      NBL LENGTH(nezev)<length;
      EFFECTS nezev' = nezev~elem;
    END insert;

    remove() -> elem;
      NBL LENGTH(nezev)>0;
      EFFECTS nezev' = REMAINDER(nezev) --
        elem = FIRST(nezev);
    END remove;

END nezev.

```

Abb. 34: Abstrakter Datentyp 'nezev' zur Beschreibung eines gemeinsamen Puffers

Element enthält. Das 'PRIO'-Konstrukt bedeutet, daß bei zwei 'insert'-Operationen der Wert des Parameters 'pr' darüber entscheidet, welche Aktivität zuerst gestartet wird. Die 'insert'-Operation hat die zwei Parameter 'pr' und 'elem'. 'pr' steuert die Reihenfolge der Einträge entsprechend der 'PRIO'-Bedingung und 'elem' bezeichnet das einzutragende Element. Die Operation ist ausführbar, wenn der Puffer seine maximale Länge noch nicht erreicht hat. Nach Ausführung der Operation ist 'elem' an die den Puffer repräsentierende Sequenz angehängt. Analog ist die 'remove'-Operation nur ausführbar, wenn der Puffer nicht leer ist. Das am längsten im Puffer befindliche Element wird entnommen.

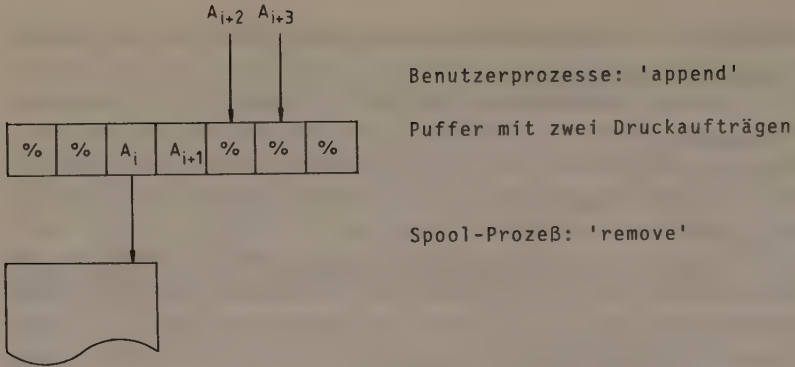


Abb. 35: Beispiel eines Spool-Puffers

3.1.2 Zusammenhang mit der Betriebssystemmaschine

Die Betriebssystemmaschine dient zur formalen Definition der Semantik der Spezifikationssprache. Dazu wird ein abstrakter Datentyp als Betriebssystemmaschine aufgefaßt. Die Interpretation der Datenbereiche wird durch die abstrakte Repräsentation und den 'PARAMETERS'-Teil definiert. Die Prozeßmenge besteht aus allen Aktivitäten, die an einem Objekt des spezifizierten Typs aktiv sind. Die globale Variablenmenge besteht aus den Komponenten der abstrakten Repräsentation. Die lokale Variablenmenge besteht aus den Operationsparametern, da diese von der jeweiligen Aktivität abhängen. Der Wertebereich der Wertfunktion w bestimmt sich aus der Typvereinbarung der Variablen.

Die Operationen der Betriebssystemmaschine entsprechen den Operationen eines abstrakten Typs. Die relationale Schreibweise

$$\sigma_{ij} = \{(d, d') \mid (a_i, d) R (a_j, d')\}$$

wird durch zwei Prädikate P und Q ersetzt:

$$\sigma_{ij} = \{(d, d') \mid P(a_i, d) \wedge Q(a_j, d, d')\}$$

Mit der Schreibweise

$$\{P(a_i, d)\} \sigma_{ij} \{Q(a_j, d, d')\}$$

wird eine Angleichung an die Hoaresche Prädikamentransformation erreicht. In der Spezifikationssprache werden die Adressen i, j durch den Operationsnamen ersetzt, und es wird der Definitionsbereich der Operation durch die Parameter eingeschränkt. Die

Hoaresche Schreibweise wird schließlich dargestellt durch

$$\begin{aligned} & \text{op_name } (p_1, \dots, p_k) \rightarrow p_{k+1}, \dots, p_n \\ & \text{PRE } P_1(ak_1, \dots, ak_m, p_1, \dots, p_k) \\ & \text{NBL } P_2(ak_1, \dots, ak_m, p_1, \dots, p_k) \\ & \text{EFFECTS } Q(ak_1, \dots, ak_m, ak'_1, \dots, ak'_m, p_1, \dots, p_n) \end{aligned}$$

Die Bedeutung der Konstrukte 'PRE', 'NBL' und 'EFFECTS' wurde bereits erklärt. Das Ausführbarkeitsprädikat der Betriebssystemmaschine wird in 'PRE'- und 'NBL'-Prädikate aufgespalten. Während durch Zustandsänderungen der abstrakten Komponenten ak_i ($1 \leq i \leq m$), die zur Erfüllung des 'NBL'-Prädikats führen, die Operation fortgesetzt werden kann, wird bei Nichterfüllung des 'PRE'-Prädikats die Operation abgebrochen. Nach Ausführung einer Operation gilt das 'EFFECTS'-Prädikat.

Die Semantik des 'COMP'- bzw. 'PRIO'-Konstrukts ist identisch mit der Bedeutung der vtgl- und prio-Relationen. Zwei Aktivitäten sind verträglich und damit parallel ausführbar, wenn das 'COMP'-Prädikat erfüllt ist. Im anderen Fall sind sie unverträglich, so daß eine Aktivität in den Zustand "wartend" versetzt wird. Den Zustand "laufend" erreicht eine Aktivität (ein Prozeß) dann, wenn keine Aktivität mit einer höheren Priorität "bereit" ist. ('PRIO'-Prädikat). Die Zustände "laufend", "bereit", "wartend" und "blockiert" sind in Kapitel 2 ausführlich dargestellt.

3.2 Die Spezifikationssprache SPAGAT

Um die vorgestellte Spezifikationssprache zur Beschreibung asynchroner, geschützter Systeme zu erweitern, genügt es prinzipiell, den Informationsfluß über die Objekte des abstrakten Typs einzuschränken. Dazu wird zunächst das 'INF'-Konstrukt eingeführt. Da ein spezielles Konstrukt für Zugriffseinschränkungen die Problembeschreibung vereinfacht, wird das 'ACC'-Konstrukt definiert. Bei der Betrachtung von Informationsflüssen sind häufig die beteiligten Prozesse von besonderem Interesse. Daraus ergeben sich Erweiterungen von SPAGAT, auf die im ersten Abschnitt kurz eingegangen wird. Die Informationsflußeinschränkungen sind nicht nur von den Operationsparametern abhängig, sondern es werden auch spezielle Objekte zur Beschreibung von Schutzproblemen verwendet (z.B. Rechte, Capabilities). Die Spezifikation solcher Objekte innerhalb eines abstrakten Typs und ihre Manipulation mit Operationen des Typs beschreibt Abschnitt zwei. Die Abschnitte drei und vier definieren das 'INF'- bzw. das 'ACC'-Konstrukt.

Im Anhang I ist eine kontextfreie Grammatik zur Beschreibung der SPAGAT-Syntax enthalten. Die Syntaxbeschreibung geht nach der in /WIRT80/ verwendeten Metasprache, einer erweiterten Backus/Naur-Form, vor. Linke und rechte Seiten der Produktionen sind durch Gleichheitszeichen getrennt, jede Produktion wird mit einem Punkt abgeschlossen. Mehrere Alternativen werden durch einen senkrechten Strich getrennt. Wiederholbare Anteile stehen in geschweiften Klammern, optionale Anteile werden von eckigen Klammern eingeschlossen. Nichtterminale Symbole werden klein geschrieben, terminale Symbole werden groß geschrieben oder in Hochkommas eingeschlossen. Zur übersichtlichen Darstellung wird auf eine ASCII-Ersatzdarstellung für Sonderzeichen verzichtet (ebenso in den Beispielen) und die bekannte mathematische Schreibweise verwendet. Geeignete Ersatzdarstellungen sind notwendig, wenn zur automatischen Bearbeitung von Typspezifikationen die Zeichensätze der verwendeten E/A-Geräte nicht ausreichen.

Auf die Syntax und Semantik der Schutzkonstrukte wird in den folgenden Abschnitten genauer eingegangen. Zur Beschreibung der Semantik werden geeignete Attribute eingeführt. Es wird keine vollständige attributierte Grammatik angegeben, da nicht die gesamte Spezifikationsprache, sondern nur die neu eingeführten Schutzkonstrukte im Vordergrund der Arbeit stehen. Die Semantik der Konstrukte selbst wird auf die Prädikate inf und acc der Betriebssystemmaschine zurückgeführt.

3.2.1 Prozeßidentifikation

Die vorgestellte Spezifikationsmethode ist auf die Spezifikation abstrakter Datentypen beschränkt. Durch die Implementationshierarchie, die über die Implementationen der abstrakten Typen hergestellt wird (siehe Kapitel 4), sind auch baumartige Systeme beschreibbar. Ein abstrakter Datentyp ist aber im wesentlichen eine passive Struktur. Die Interpretation der Aktivitäten eines abstrakten Typs als Prozesse einer Betriebssystemmaschine entspricht nur der Tatsache, daß die Operationen an einem abstrakten Typ unabhängig (bis auf Parameter) vom umgebenden System, in dem diese Operationen aufgerufen werden, synchronisiert werden können.

Obwohl abstrakte Datentypen den modularen Aufbau von Software-Systemen unterstützen, da sie die Definition hierarchischer Teilsysteme erlauben und einen Abstraktionsmechanismus enthalten, sind zum Aufbau von Software-Systemen konkrete Objekte und Subjekte notwendig. Im Gegensatz zu SPAGAT, wo nur Parameter spezifiziert werden, müßte eine Sprache zur Spezifikation asynchroner, geschützter Systeme Deklarationen zur Erzeugung der Objekte des Systems und ein Prozeßkonzept zur Erzeugung der Subjekte des Systems enthalten. Auch diese Sprache ließe sich aus der Betriebssystemmaschine durch eine geeignete Interpretation ($\text{Subjektmenge} = \bar{P}$, $\text{Objektmenge} = \text{Var}$) ableiten.

Den Zusammenhang zwischen abstrakten Typen und darauf aufbauenden Systemen zeigt Bild 36. Grundlegend ist eine Menge von

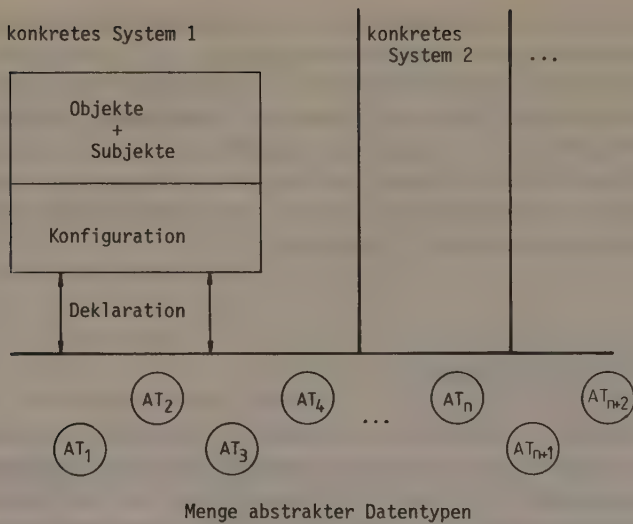


Abb. 36: Zusammenhang zwischen abstrakten Typen und darauf aufbauenden Systemen

abstrakten Datentypen, die als Bausteine des zu spezifizierenden Systems angesehen werden sollen. Zur Spezifikation von Subjekten ist ein Prozeßkonzept einzuführen. Bei der Erzeugung eines Objekts dieses Typs werden gleichzeitig alle mit dem Objekt verbundenen "Prozesse" erzeugt. Mit Hilfe der abstrakten Typen werden bei Realisierung der Systemspezifikation Objekte und Subjekte generiert und den Grundbausteinen der implementierenden Datentypen Repräsentationen auf einer konkreten Maschine zugeordnet. Konzepte zur Systemspezifikation werden nicht weiter ausgearbeitet, da sie nicht zum Thema der vorliegenden Arbeit gehören.

Die Bezüge zwischen abstrakten Datentypen und konkreten Software-Systemen treten besonders bei Schutzaspekten in Erscheinung. Ein abstrakter Datentyp stellt nur ein Schema zur Deklaration von Objekten dar. Bei der Spezifikation von Synchronisationsproblemen ist das Schema meist ausreichend, da die Synchronisation von Operationen eines abstrakten Typs unabhängig von den aufrufenden Prozessen erfolgt. Diese Eigenschaft gilt

für Schutzprobleme i.allg. nicht. Z.B. sollen Informationsflüsse nicht allgemein beschränkt werden, sondern nur bei bestimmten aufrufenden Prozessen. In Capability-Systemen sind Rechte, die zur Ausführung einer Operation berechtigen, nicht den Operationen oder Aktivitäten fest zugeordnet, sondern hängen vom Aufrufer ab. Deshalb müssen zur Formulierung von Informationsflußbedingungen für Aktivitäten die aufrufenden Prozesse identifizierbar sein.

Zur syntaktischen Repräsentation des Identifikators gibt es zwei Möglichkeiten:

- Der Identifikator wird explizit in die Parameterliste der Operationen aufgenommen.
- Der Identifikator wird implizit der Aktivität zugeordnet, die beim Operationsaufruf entsteht.

Die erste Möglichkeit vereinfacht die Modellierung der Schutzaussagen, da auf explizit definierte Objekte zurückgegriffen werden kann. Die verwendeten Parameternamen und ihre Werte, ebenso der Prozeßidentifikator, sind mit dem bestehenden Spezifikationskonzept zu vereinbaren. Nachteilig ist, daß damit die spezifizierten Schutzeinschränkungen umgangen werden können, da z.B. der Prozeßname für den aufrufenden Prozeß über den Parameterübergabemechanismus sichtbar ist. Die Schutzeinschränkungen sind nicht mehr auf den abstrakten Datentyp konzentriert.

Die zweite Möglichkeit hat den Nachteil, daß spezielle Konstrukte zum Zugriff auf die Identifikation eines Prozesses in SPAGAT aufgenommen werden müssen. Vorteilhaft ist, daß die Zuordnung zwischen aufrufendem Prozeß und dem Identifikator nicht sichtbar ist, sondern implizit erfolgt.

In der vorliegenden Arbeit soll ein Kompromiß aus beiden Möglichkeiten verwendet werden. Der Identifikator des aufrufenden Prozesses oder in Schutzbedingungen verwendete Objekte werden als Parameter übergeben. Die Parameter werden aber speziell gekennzeichnet, so daß ihre besondere Verwendung als Schutzparameter deutlich wird. Konsequenzen für die Implementation werden in Kapitel 4 behandelt.

3.2.2 Schutzobjekte

Alle Objekte, die in Schutzbedingungen vorkommen, müssen gekennzeichnet sein. Dies geschieht im Teil 'PROTECTION', wo die entsprechenden Parameter hinter dem reservierten Wort 'PROT' aufgeführt werden müssen (siehe Abb.37).

```
schutzteil = [ PROTECTION
               {schutzobjekte ";"}
               [informationsflussbeschaenkung ";"] ] .
```

```
schutzobjekte = PROT identifikator {"," identifikator} .
               identifikator.prot = TRUE;
```

Abb. 37: Der Teil 'PROTECTION'

Die Kennzeichnung eines Parameters einer Operation als Schutzparameter hängt vom jeweiligen Problem ab. Wird in einem abstrakten Typ die Prozeßpriorität nur zur Synchronisation verwendet, entfällt das 'PROT'-Attribut. Geht die Priorität in Informationsflußbeschränkungen ein, muß der Prioritätsparameter das Attribut 'PROT' erhalten (auch wenn die Priorität zusätzlich zur Synchronisation verwendet wird).

Zur Formulierung von Schutzproblemen, insbesondere zur Formulierung von Zugriffsschutzeinschränkungen, reichen 'PROT'-Parameter nicht aus, wenn nicht die problemnahe Formulierbarkeit von Schutzproblemen eingeschränkt werden soll. Ein Beispiel sind Schlüsselsysteme, wo am Objekt eine Menge von Schlössern hinterlegt ist. Ein Objektzugriff ist dann erlaubt, wenn ein übergebener Schlüssel (als 'PROT'-Parameter) in eines der Schlösser paßt.

Deshalb sollen nicht nur Operationsparameter als geschützte Objekte eines abstrakten Typs spezifiziert werden können, sondern auch Komponenten der abstrakten Repräsentation. Als Identifikatoren der zweiten Produktion in Bild 37 sind also Parameter und abstrakte Komponenten erlaubt. Mit 'PROT' gekennzeichnete Komponenten heißen auch Schutzkomponenten.

Schutzobjekte unterscheiden sich bez. ihrer Verwendung innerhalb eines abstrakten Typs nicht von anderen Objekten (Parametern, Komponenten), außer wenn sie in Schutzbedingungen vorkommen. U.U. können eigene Operationen zur Manipulation der Schutzkomponenten eingeführt werden. Z.B. können in einem Schössersystem Schlösser hinzugefügt oder entfernt werden. Diese Operationen (Schutzoperationen) erfordern selbst Schutzmaßnahmen. Diese sind ebenfalls mit Hilfe der Schutzkonstrukte beschreibbar. Z.B. könnte ein spezielles Schloß für die Schloßänderungsfunktionen definiert werden. Im bereits mehrfach erwähnten "Niedrigwasser"-Mechanismus sind die Klassifikationsänderungen (die Klassifikation kann als Schutzobjekt aufgefaßt werden) in den übrigen Operationen enthalten.

Die Integration der Schutzobjekte in den abstrakten Typ impliziert ihre Berücksichtigung in der Synchronisation (siehe auch 3.2.4: Zusammenhang zwischen 'ACC'- und 'NBL'-Konstrukt). Die spezifizierten Synchronisationsbedingungen müssen garantieren, daß das Schutzobjekt stets in einem konsistenten Zustand ist. Zu bemerken ist, daß z.B. beim Rückruf von Capabilities das Problem entsteht, wann ein Zugriff ungültig gemacht werden soll. Mit Hilfe der 'PRIO'-Bedingung kann z.B. spezifiziert werden, daß der Rückruf erst wirksam wird, wenn es keine anderen Aktivitäten mehr am betrachteten abstrakten Objekt gibt.

3.2.3 Das 'INF'-Konstrukt

Zunächst sollen anhand der Tabelle in Abb. 38 die möglichen Informationsflüsse an einem abstrakten Objekt erläutert werden. A_1 und A_2 seien Aktivitäten und B eine Bedingung abhängig von den Operationsparametern und dem Zustand des abstrakten Objekts. Fall 2 beschreibt den Informationsfluß zwischen zwei Aktivitäten A_1 und A_2 . In Fall 1 wird der Informationsfluß zusätzlich durch eine Bedingung B eingeschränkt. In den Fällen 3 - 6 wird nur die Informationsquelle bzw. -senke betrachtet. Auch hier sind die Informationsflüsse wieder durch Bedingungen einschränkbar. Im Fall 7 wird, unabhängig von der Aktivität,

	A ₁	A ₂	B	
1	A ₁	A ₂	B	direkter Informationsfluß von A ₁ nach A ₂ wenn B
2	A ₁	A ₂	TRUE	direkter Informationsfluß von A ₁ nach A ₂
3	A ₁	?	B	direkter Informationsfluß von A ₁ falls B
4	A ₁	?	TRUE	direkter Informationsfluß von A ₁
5	?	A ₂	B	direkter Informationsfluß nach A ₂ wenn B
6	?	A ₂	TRUE	direkter Informationsfluß nach A ₂
7	?	?	B	direkter Informationsfluß falls B
8	?	?	TRUE	direkter Informationsfluß

Abb. 38: Klassifikation von Informationsflüssen

nur die angegebene Bedingung B untersucht, während Fall 8 alle möglichen Informationsflüsse zuläßt. Praktische Beispiele zeigen, daß Fall 1 besonders interessant ist, wenn der Informationsfluß zwischen zwei Aktivitäten A₁ und A₂ der Bedingung B unterworfen wird. So wird z.B. im "Niedrigwasser"-Mechanismus (/CHEH81/) gefordert, daß Informationsfluß zwischen zwei Aktivitäten genau dann erlaubt ist, wenn die lesende Aktivität eine höhere Klassifikationsnummer als die schreibende hat (Informationsfluß von "unten" nach "oben").

Die Tabelle zeigt, daß das in Kapitel 2 eingeführte inf-Prädikat die Formulierung der verschiedenen Informationsflußmöglichkeiten erlaubt. Deshalb wird ein 'INF'-Konstrukt eingeführt mit der in Abb.39 enthaltenen Syntax. Eine Informationsfluß einschränkung wird eingeleitet durch die 'INF'-Funktion (mit zwei Aktivitätsvariablen als Parametern). Der Äquivalenzpfeil '<=>' soll andeuten, daß Information (von der ersten zur zweiten Aktivität) genau dann fließen kann, wenn der nachfolgende prädikatenlogische Ausdruck erfüllt ist. Werden keine Aktivitätspa-

parameter angegeben, so ist der Informationsfluß zwischen beliebigen Aktivitäten eines abstrakten Objekts zu untersuchen (Fall 7 der Tabelle). Da das 'INF'-Konstrukt nicht speziellen Operationen zugeordnet ist, wird es im bereits eingeführten Teil 'PROTECTION' spezifiziert (siehe Syntax des 'PROTECTION'-Teils).

```

informationsflussbeschaenkung =
    inf_funktion "<=>" praedikatenlogischer_ausdruck .
    (∃v∈praedikatenlogischer_ausdruck.var)
    ((v.typ=ACTIVITY ∧ v∈inf_funktion.var)
    v (v.typ#ACTIVITY ∧ ¬v.prot) ----> FEHLER);
informationsflussbeschaenkung.wert =
    praedikatenlogischer_ausdruck.wert;

inf_funktion = (INF|NINF) ["("identifikator(1),identifikator(2))"].
    identifikator(1).typ#ACTIVITY v identifikator(2).typ#ACTIVITY
    ----> FEHLER;
inf_funktion.var = identifikator(1).var U identifikator(2).var;
    
```

Abb. 39: Informationsflußeinschränkungen eines abstrakten Typs

Zur Definition der Semantik werden verschiedene Attribute eingeführt (/WILH79/). Das Attribut 'wert' enthält den Wert eines Ausdrucks. Mit 'typ' wird der Typ einer Variablen bestimmt, während 'prot'='TRUE' eine Variable als geschützt kennzeichnet ('FALSE' sonst). Das Attribut 'var' enthält die Identifikatoren aller freien Variablen eines Ausdrucks. Frei sollen alle abstrakten Komponenten und die Parameter der sichtbaren Operationen heißen. Es wird nicht zwischen Attributen der statischen und der dynamischen Semantik unterschieden. Ziffern in runden Klammern unterscheiden nichtterminale Symbole gleichen Namens in einer Produktion. Ziffern in spitzen Klammern kennzeichnen Alternativen der rechten Seite einer Produktion.

Der prädikatenlogische Ausdruck der 'INF'-Bedingung präzisiert, analog zu den Synchronisationsbedingungen 'COMP' und 'PRIO', die Aktivitäten, für die die 'INF'-Bedingung gilt, und zusätzliche Bedingungen für den Informationsfluß, abhängig von Schutzparametern und dem Zustand des abstrakten Objekts. Die Bedeutung einer 'INF'-Bedingung wird mit Hilfe des Modells der Betriebssystemmaschine definiert:

$$\text{informationsflussbeschaenkung.wert} \text{ <---> } \text{inf}(\Sigma_1, \Sigma_2, o, B_1, B_2)$$

Die Operationsmengen Σ_1 und Σ_2 ergeben sich aus den Spezifikationen für die Aktivitäten in 'prädikatenlogischer_ausdruck'. Die Variable o ist identisch mit dem abstrakten Objekt 'ao', das alle betrachteten Operationen gemeinsam haben (siehe 3.1). Die Bedingung $B_1 \wedge B_2$ schließlich ist erfüllt, wenn "prädikatenlogischer_ausdruck" erfüllt ist. Bei Angabe einer 'NINF'-Bedingung gilt:

informationsflussbeschaenkung.wert <---> $\neg \text{inf}(\Sigma_1, \Sigma_2, o, B_1, B_2)$

Das bedeutet, daß die an einem abstrakten Datentyp möglichen Informationsflüsse durch die Bedingung des 'INF'-Konstrukts ('INF'-Bedingung) bestimmt werden. Das Fehlen einer 'INF'-Bedingung bedeutet, daß direkte Informationsflüsse über Objekte eines abstrakten Typs (abstrakte Objekte) nicht eingeschränkt sind. Dies ist äquivalent mit der konstanten 'INF'-Bedingung 'TRUE'. Umgekehrt führt die konstante 'INF'-Bedingung 'FALSE' dazu, daß Objekte des abstrakten Typs nur als Informationsquellen oder -senken benutzt werden können. Da eine 'INF'-Bedingung auch als Einschränkung der Informationsflüsse eines abstrakten Typs aufgefaßt werden kann, spricht man auch von Informationsflußbeschränkungen.

Die Allgemeinheit des inf-Prädikats konnte, im vorliegenden 'INF'-Konstrukt beibehalten werden. Einschränkungen der Operationenmenge werden in den prädikatenlogischen Ausdrücken angegeben. Das in Bild 38 enthaltene Fragezeichen ("?"), das die gesamte Operationenmenge Σ der Betriebssystemmaschine repräsentiert, wird berücksichtigt, wenn die 'INF'-Bedingung keine Aktivitätsparameter enthält. Auf diese Weise werden Einschränkungen für alle Operationen eines abstrakten Typs definiert, da mit jeder Operation Informationsflüsse verbunden sind (außer es handelt sich um die Spezifikation einer Informationsquelle oder -senke).

Besonders zu berücksichtigen ist der Informationsfluß nach Objektinitialisierung, da dadurch, daß mögliche Änderungen nicht oder bereits durchgeführt sind, Information fließen kann. Um solche Informationsflüsse zu spezifizieren, wird die für einen abstrakten Datentyp konstante Aktivität 'INIT_ACTIVITY'

eingeführt. Wird in einer 'INF'-Bedingung 'INIT_ACTIVITY' als Informationsquelle verwendet (als Informationssenke ist es sinnlos), so bedeutet dies, daß mögliche Informationsflüsse, die sich aus dem Initialzustand des abstrakten Objekts ergeben, erlaubt sind.

3.2.4 Zugriffseinschränkungen

Obwohl Informationsflußeinschränkungen zur Formulierung von Schutzproblemen ausreichen, soll ein weiteres Schutzkonstrukt eingeführt werden, das die Formulierung von Einschränkungen bez. einzelner Operationen vereinfacht. Obwohl mit dem 'INF'-Konstrukt diese Einschränkungen auch zentral formulierbar sind, vereinfacht die Zuordnung der Einschränkungen zu den betreffenden Operationen die Bedingungen und macht sie dadurch übersichtlicher. Da die Zuordnung der Einschränkungen zu einzelnen Operationen den Informationsflußaspekt ausschließt (dazu wären eine ändernde und eine abfragende Operation nötig), soll zur semantischen Definition das acc-Prädikat der Betriebssystemmaschine verwendet werden.

Die Syntax des Konstrukts beschreibt Abb.40. Die erste Produktion enthält die möglichen einschränkenden Bedingungen einer Operation. Ist eine 'PRE'-Bedingung nicht erfüllt, wird die Operation abgebrochen, ist eine 'NBL'-Bedingung nicht erfüllt, wird die Operation blockiert. Die vierte Produktion erlaubt als Bedingungen allgemeine prädikatenlogische Ausdrücke oder spezielle Schutzausdrücke. Diese werden durch das reservierte Wort 'ACC' eingeleitet, das eine boolesche Funktion ohne Parameter repräsentiert. Schutzausdrücke dürfen nur geschützte Objekte des abstrakten Typs enthalten.

Der Wert einer 'ACC'-Bedingung (Zugriffseinschränkung) wird genau dann wahr, wenn das zugehörige acc-Prädikat der Betriebssystemmaschine erfüllt ist. Daraus folgt die Ausführbarkeit der Operation, wenn nicht noch zusätzliche Informationsflußeinschränkungen ihre Ausführung verbieten.

praedikatenlogischer_ausdruck(2).wert <--> acc(E,o,B)

```
operationseinschraenkung = [ pre_bedingung ] [ nbl_bedingung ] .
```

```
pre_bedingung = PRE { vorbedingung "--" } .  
pre_bedingung.wert=vorbedingung.wert;
```

```
nbl_bedingung = NBL { vorbedingung "--" } .  
nbl_bedingung.wert = vorbedingung.wert;
```

```
vorbedingung = praedikatenlogischer_ausdruck(1) <1> |  
                ACC "<=>" praedikatenlogischer_ausdruck(2) <2> .  
<1> vorbedingung.wert = praedikatenlogischer_ausdruck(1).wert;  
<2> vorbedingung.wert = praedikatenlogischer_ausdruck(2).wert;  
<2> (( $\exists v \in$  praedikatenlogischer_ausdruck(2).var) (  $\neg v$ .prot  
                ---> FEHLER));
```

Abb. 40: Zugriffseinschränkungen eines abstrakten Typs

Die Variable o wird durch das abstrakte Objekt 'ao' bestimmt, das für jede Operation bekannt ist. Die Operationenmenge Σ ist durch die Typoperation bestimmt, der die 'ACC'-Bedingung zugeordnet ist. Der Wert der Zugriffseinschränkung reduziert sich somit auf den Wert der Bedingung B, d.h.

```
praedikatenlogischer_ausdruck(2).wert <---> B
```

Obwohl das 'ACC'-Konstrukt neben den bereits vorhandenen Konstrukten 'PRE' und 'NBL' einzuordnen ist, soll es den beiden untergeordnet werden, d.h., es kann als Bedingung im 'PRE'- oder im 'NBL'-Teil einer Operation auftreten. Damit kommt zum Ausdruck, daß die Nichterfüllung der 'ACC'-Bedingung, abhängig vom umgebenden Konstrukt, entweder zum Operationsabbruch führt ('PRE') oder zur Blockierung ('NBL'). Wird die Operation aufgrund des 'ACC'-Konstrukts blockiert, können Änderungen des Werts der 'ACC'-Bedingung zur Fortsetzung der Operation führen. Statt von Operationsabbruch sollte man vom Nichtbeginn einer Operation sprechen, falls die 'PRE'-Bedingung nicht erfüllt ist. Dadurch werden die Konsistenzprüfungen vor Operationsbeginn von der eigentlichen Operation getrennt.

Mit der Zuordnung des 'ACC'-Konstrukts sowohl zum 'PRE'- als auch zum 'NBL'-Teil wird der Zusammenhang zwischen Schutzeinschränkungen und Synchronisationseinschränkungen, wie er aus Kapitel 1.1 erwartet wird, deutlich. Ein praktisches Beispiel

ist, daß besonders schutzwürdige Gebäude (z.B. SIEMENS-Forschungszentrum) häufig nur in Begleitung autorisierter Personen betreten werden dürfen. Der Zutritt ist zunächst ein Schutzproblem. Fehlt beim Ankommen eines Besuchers die begleitende Person, so wird der Besuch nicht abgebrochen, sondern der Besucher wartet auf die Person, mit der er das Gebäude betreten darf (Synchronisationsproblem). Gibt es keine solche Person, darf das Gebäude nicht betreten werden.

Ein weiterer Aspekt des 'ACC'-Konstrukts ist die Zuordnung einer 'ACC'-Bedingung zum 'PRE'- oder zum 'NBL'-Teil, die i.allg. von der Problemstellung abhängt. Die Zuordnung gehört zur Schutzstrategie. Am Beispiel: Man kann Besucher, für die bei der Ankunft keine Begleitperson zur Verfügung steht, entweder fortschicken oder warten lassen.

3.3 Beispiele

Nachdem im vorigen Abschnitt die zur Formulierung von Schutzproblemen notwendigen Konstrukte definiert wurden, sollen im weiteren einige ausführliche Beispielspezifikationen ausgeführt werden, um die Verwendung der Konstrukte zu verdeutlichen.

3.3.1 N-Erzeuger-1-Verbraucher-Problem

Das N-Erzeuger-1-Verbraucher-Problem ('nezev') wurde bereits in Abschnitt 3.1 vorgestellt. Der folgende abstrakte Typ 'pnezev' (p: protected) soll zusätzlich Informationsflußkontrollen auf den Elementen eines Puffers vom Typ 'pnezev' spezifizieren. Der Puffer soll die Eigenschaft haben, daß die über den Puffer möglichen Informationsflüsse zwischen den Operationen 'insert' und 'remove' das militärische Sicherheitsmodell erfüllen. Bild 41 zeigt die Spezifikation des abstrakten

Datentyps 'pnezev'. Das Schutzproblem wird als 'NINF'-Bedingung formuliert. Die Bedingung besagt, daß zwischen zwei Aktivitäten 'a1' und 'a2' keine Information fließen darf, wenn die 'remove'-Aktivität eine kleinere Klassifikation 'prot_class' als die 'insert'-Aktivität hat, die das zu entfernende Element eingetragen hat. Die Implementation der Konstrukte in Kapitel 4 macht deutlich, warum das 'NINF'-Konstrukt verwendet wird (versteckte Kanäle). Spezifikationssprachlich ist die Bedingung

$$\text{INF}(a1,a2) \Leftrightarrow a1 \in \text{insert}(\text{prot_class}) \wedge a2 \in \text{remove}(\text{prot_class}) \\ \wedge a1.\text{prot_class} \leq a2.\text{prot_class}$$

äquivalent.

ABSTRACT TYPE SPECIFICATION pnezev

(t: TYPE; max_elements: CARDINAL WHERE max_elements > 0);

ABSTRACT REPRESENTATION

pnezev: SEQUENCE OF t INIT EMPTY_SEQUENCE;

PARAMETERS

prot_class: (unbeschraenkt, vertraulich, geheim, streng_geheim);
elem: t;
a1, a2: ACTIVITY;

SYN

COMP(a1,a2) $\Leftrightarrow$ (a1 $\in$ insert $\wedge$ a2 $\in$ remove) $\wedge$ (LENGTH(pnezev) > 1);

PROTECTION

PROT prot_class;
NINF(a1,a2) $\Leftrightarrow$ (a1 $\in$ insert(prot_class) $\wedge$ a2 $\in$ remove(prot_class2) $\wedge$ (a1.prot_class > a2.prot_class);

OPERATIONS

insert(prot_class, elem);
NBL LENGTH(pnezev) < max_elements;
EFFECTS pnezev' = pnezev ~ elem;
END insert;

remove(prot_class) -> elem;
NBL LENGTH(pnezev) > 0;
EFFECTS pnezev' = REMAINDER(pnezev) --
elem = FIRST(pnezev);
END remove;

END pnezev.

Abb. 41: Abstrakter Typ 'pnezev'

Aus der Spezifikation des Schutzproblems (zwischen je zwei Paaren 'insert'/'remove' darf Information nur von einem niedriger klassifizierten zu einem höher klassifizierten Prozeß fließen) ergeben sich Auswirkungen auf andere Spezifikations-teile. So ist im Teil 'PARAMETERS' der 'prot_class'-Parameter

zu spezifizieren, der in den Operationen 'insert' und 'remove' verwendet wird. Der 'prot_class'-Parameter soll die Klassifikation des aufrufenden Prozesses enthalten. Die Parameterposition erlaubt aber beliebige Interpretationen und macht nur deutlich, daß der Parameter für die Überprüfung in Schutzkonstrukten auftreten darf. Zur Verwendung in der 'NINF'-Bedingung ist 'prot_class' als geschützt zu kennzeichnen.

Als nächstes soll ein abstrakter Datentyp 'apnezev' (access-protected) spezifiziert werden, bei dem nur die Zugriffe auf die Operationen des abstrakten Datentyps eingeschränkt werden (siehe Bild 42). Jeweils ein Pufferelement soll nur gelesen werden dürfen, wenn die Klassifikation des lesenden Prozesses größer oder gleich der Klassifikation des eintragenden Prozesses ist. Der Unterschied zum Typ 'pnezev' besteht darin, daß nicht Informationsflüsse zwischen Aktivitäten, sondern Zugriffe auf das abstrakte Objekt eingeschränkt werden. Da im vorliegenden Fall nur der Ausgabeparameter 'elem' existiert, sind die Schutzeinschränkungen in 'pnezev' und 'apnezev' äquivalent. Unterschiede werden in Kapitel 4 deutlich, da in der Implementation zusätzliche Ausgabeparameter berücksichtigt werden müssen.

Zur Realisierung dieser Strategie ist im Teil 'PROTECTION' eine Schutzkomponente zu definieren. Da bei jeder 'remove'-Operation die Klasse des eintragenden Prozesses überprüft werden muß, ist die abstrakte Repräsentation um eine Komponente zu erweitern, die jedem Pufferelement die Klasse des eintragenden Prozesses zuordnet. Die Zuordnung geschieht über die Reihenfolge der Sequenzeinträge in der Komponente 'prot_data' (siehe Abb.42).

In den Typoperationen müssen die Zugriffsprüfungen mit Hilfe des 'ACC'-Konstrukts eingefügt werden. Dabei kann die fehlende Zugriffserlaubnis (Klasse des Pufferelements größer als Klasse des entfernenden Prozesses) entweder zum Blockieren führen oder zum Operationsabbruch. Beim Blockieren wird dann solange gewartet, bis ein Element mit einer Klasse als erstes im Puffer steht, die kleiner oder gleich der des blockierten Prozesses ist. Zu bemerken ist, daß die Operationen auch die 'prot_data'-

```

ABSTRACT TYPE SPECIFICATION apnezev
  (t: TYPE; max_elements: CARDINAL WHERE max_elements>0);

ABSTRACT REPRESENTATION
  apnezev: SEQUENCE OF t INIT EMPTY_SEQUENCE;
  prot_data: SEQUENCE OF class INIT EMPTY_SEQUENCE;

TYPES
  class = (unbeschraenkt, vertraulich, geheim, streng_geheim);

PARAMETERS
  prot_class: class;
  elem: t;
  a1, a2: ACTIVITY;

SYN
  COMP(a1, a2) <=> (a1 ∈ insert ∧ a2 ∈ remove) ∧ (LENGTH(apnezev)>1);

PROTECTION
  PROT prot_class, prot_data;

OPERATIONS
  insert(prot_class, elem);
  NBL LENGTH(apnezev) < max_elements;
  EFFECTS apnezev' = apnezev ~ elem --
          prot_data' = prot_data ~ prot_class;
  END insert;

  remove(prot_class) -> elem;
  PRE ACC <=> FIRST(prot_data) ≤ prot_class;
  NBL LENGTH(apnezev) > 0;
  EFFECTS apnezev' = REMAINDER(apnezev) --
          prot_data' = REMAINDER(prot_data) --
          elem = FIRST(apnezev);
  END remove;

END apnezev.

```

Abb. 42: Abstrakter Typ 'apnezev'

-Komponente der abstrakten Repräsentation ändern, um die Klasse des eintragenden Prozesses dem jeweiligen Pufferelement zuzuordnen. Das Ergebnis für den Operationsteil zeigt ebenfalls Bild 42, wo die Operation 'remove' eine 'ACC'-Bedingung enthält.

Nachdem die bisherigen Beispiele fest strukturierte ("mandatory", siehe Kapitel 1.1) Schutzprobleme beschrieben haben, soll im Typ 'rnezev' (r: rights) ein beliebig strukturiertes ("discretionary") Schutzproblem mit Rechten definiert werden (siehe Bild 43). Über die Verteilung der Rechte werden im Bei-

spiel keine Angaben gemacht. Jeder Operation des abstrakten Datentyps wird ein Recht ('Insert', 'Remove') zugeordnet. Außerdem gibt es für jede Operation einen Parameter, der die Rechte des aufrufenden Prozesses enthält. Beim Operationsaufruf muß das zugeordnete Recht in der übergebenen Rechtemenge enthalten sein (siehe 'ACC'-Bedingungen).

```

ABSTRACT TYPE SPECIFICATION rnezev
    (t: TYPE; max_elements: CARDINAL WHERE max_elements>0);

ABSTRACT REPRESENTATION
    rnezev: SEQUENCE OF t INIT EMPTY_SEQUENCE;

TYPES
    rights = (Insert, Remove);

PARAMETERS
    prot_r: SET OF rights;
    elem: t;
    a1,a2: ACTIVITY;

SYN
    COMP(a1,a2) <=> (a1∈insert ∧ a2∈remove) ∧ (LENGTH(rnezev)>1);

PROTECTION
    PROT prot_r;

OPERATIONS
    insert(prot_r,elem);
        PRE ACC <=> Insert∈prot_r;
        NBL LENGTH(rnezev)<max_elements;
        EFFECTS rnezev'= rnezev~elem;
    END insert;

    remove(prot_r) -> elem;
        PRE ACC <=> Remove∈prot_r;
        NBL LENGTH(rnezev)>0;
        EFFECTS rnezev'= REMAINDER(rnezev) --
            elem=FIRST(rnezev);
    END remove;

END rnezev.

```

Abb. 43: Abstrakter Typ 'rnezev'

Im abstrakten Typ 'rnezev' wird die 'ACC'-Bedingung dem 'PRE'-Teil einer Operation zugeordnet, d.h., ist das Recht nicht vorhanden, wird die Operation nicht ausgeführt. In Abschnitt 3.4 wird auf die Möglichkeit hingewiesen, den Zugriff aufgrund einer nichterfüllten 'ACC'-Bedingung nicht zu unterbinden, sondern zu blockieren. Dazu müßte die 'ACC'-Bedingung

dem 'NBL'-Teil zugeordnet werden. Diese einfache Umformung führt im obigen Beispiel zu einem Widerspruch, da Parameterwerte einer Aktivität sich nicht ändern können, sondern nur der Zustand eines abstrakten Objekts. Um deshalb die beschriebene Wartestrategie formulieren zu können, müßte die Zuordnung von Rechten zu Subjekten in einer Schutzkomponente des abstrakten Typs hinterlegt werden und eine Operation zur Änderung des Schutzobjekts definiert werden. Diese Strategie wird am Beispiel des Typs 'dnezev' (d: distribute, Bild 44) gezeigt, wo die abstrakte Komponente 'users' die Rechte der aufrufenden Prozesse enthält. Zur Verwendung in Schutzbedingungen wird 'users' als geschützt gekennzeichnet. Mit der Operation 'distribute' können einem Benutzer Rechte zugeordnet werden. An diesem Beispiel wird gleichzeitig die Verteilung der Rechte auf verschiedene Subjekte sichtbar. Die Verteiloperation selbst muß auch geschützt werden. Um das dafür notwendige Recht und seine Verteilung nicht erneut kontrollieren zu müssen, wird die Verteilung einem speziellen Prozeß zugeordnet, der bei Objektinitialisierung vorgegeben wird. Dazu wird in 'dnezev' ein spezieller Identifikator 'distributor' verwendet. Beim Aufruf der Operation 'distribute' wird überprüft, ob der aufrufende Prozeß 'prot_id' identisch mit 'distributor' ist, da nur dann die Operation ausführbar ist.

```
ABSTRACT TYPE SPECIFICATION dnezev (t:TYPE;
    d:ID; max_elements:CARDINAL WHERE max_elements>0);
```

ABSTRACT REPRESENTATION

```
dnezev: SEQUENCE OF t INIT EMPTY_SEQUENCE;
users: SET OF RECORD r: SET OF rights; id: ID END
    INIT {};
distributor: ID INIT d;
```

TYPES

```
rights = SET OF (Insert, Remove);
ID = ARRAY[1..8] OF CHAR;
```

PARAMETERS

```
prot_id, id_p, id: ID;
elem: t;
R_p, R: rights;
a1,a2: ACTIVITY;
```

SYN

```
COMP(a1,a2) <=> (a1∈insert ∧ a2∈remove) ∧ (LENGTH(dnezev)>1);
```

PROTECTION

```
PROT users,distributor,prot_id;
```

OPERATIONS

```
insert(prot_id,elem);
    NBL ACC  $\bar{\Leftarrow}$  => ( $\exists(R,id) \in users$ )(id=prot_id ∧ Insert ∈ R) --
        LENGTH(dnezev) < max_elements;
    EFFECTS dnezev' = dnezev ~ elem;
END insert;

remove(prot_id) -> elem;
    NBL ACC  $\bar{\Leftarrow}$  => ( $\exists(R,id) \in users$ )(id=prot_id ∧ Remove ∈ R) --
        LENGTH(dnezev) > 0;
    EFFECTS dnezev' = REMAINDER(dnezev) --
        elem = FIRST(dnezev);
END remove;

distribute(prot_id,R_p,id_p);
    PRE ACC <=> prot_id=distributor;
    EFFECTS ( $\exists(R,id) \in users$ ')(R=R_p ∧ id=id_p);
END distribute;
```

END dnezev.

Abb. 44: Abstrakter Typ 'dnezev'

3.3.2 Datenschutz

In einem Industriebetrieb ist die Einführung eines Personalinformationssystems geplant. Gleichzeitig plant die Fertigungsabteilung ein System zur Produktionsdatenerfassung, das auch Daten für die Lohnbuchhaltung erfaßt. Beide Systeme sollen nicht nebeneinander auf verschiedenen Anlagen laufen, sondern es soll ein Gesamtkonzept für ein integriertes System entwickelt werden, das auf einer gemeinsamen Personaldatenbank aufbaut. Dieses Konzept stößt bei den Mitarbeitern auf Widerstand, da mit Recht auf die gravierenden Mißbrauchsmöglichkeiten (personenbezogene Arbeitsstatistik, Überwachung) hingewiesen wird. Daraufhin kommt es zu einer Betriebsvereinbarung, in der die Nutzungsmöglichkeiten des integrierten Systems den jeweiligen Bedürfnissen der verschiedenen Abteilungen angepaßt werden. Damit wird dem Personalbüro der Zugriff auf Adresse, Name, Position, Lohngruppe, Sozialdaten erlaubt. Die Lohnbuchhaltung erhält Zugriff auf Name, Lohnabrechnung, Arbeitszeit, Steuerkarte und Sozialnummer, während die Produktionsabteilung lediglich auf Arbeitsdaten und Namen der Mitarbeiter Zugriff hat.

Im folgenden wird der abstrakte Datentyp 'Produktion' spezifiziert, der die Schnittstelle zwischen Produktionsabteilung und Personaldatenbank festlegt. Analog ist für die Abteilungen Lohnbuchhaltung und Personal vorzugehen. Die abstrakten Datentypen können als Sichten der einzelnen Abteilungen auf die Datenbank interpretiert werden. Die Entwicklung des konzeptionellen Schemas der Datenbank wird von den abstrakten Typen nicht beeinflußt. Die Operationen der Datentypen können als Basis für Programme verwendet werden, die in den jeweiligen Abteilungen benötigt werden.

Zur Durchsetzung der Vereinbarung wird die Schnittstelle zur Personaldatenbank spezifiziert und mit den oben angegebenen Operationseinschränkungen versehen. Der abstrakte Typ ist nicht vollständig spezifiziert, die Operationen zeigen aber, daß die in der Betriebsvereinbarung vorgesehenen Operationseinschränkungen in Abhängigkeit von der Abteilung formal wiedergegeben

ABSTRACT TYPE SPECIFICATION Produktion (Leiter: uid);

ABSTRACT REPRESENTATION

Produktion: SET OF

RECORD Name, Arbeitsdaten: ARRAY OF CHAR END

INIT {};

leiter: uid INIT Leiter;

special: SET OF uid INIT {};

TYPES

uid = ARRAY OF CHAR;

PARAMETERS

datenmenge: SET OF ARRAY OF CHAR;

d,n,name,adaten: ARRAY OF CHAR;

prot_id, id: uid;

SYN

NCOMP(a1,a2) <=> a1∈put_Proddaten ∧

a2∈read_Proddaten ∨ get_Proddaten ∨ give_special_right

∨ a1∈get_Proddaten ∧ a2∈give_special_right;

PROTECTION

PROT leiter, special, prot_id;

OPERATIONS

read_Proddaten() -> datenmenge;

EFFECTS datenmenge = {d | (∃n)((n,d)∈Produktion)};

END read_Proddaten;

get_Proddaten(prot_id,name) -> adaten;

NBL ACC <=> prot_id∈special ∨ prot_id=leiter;

EFFECTS (∃n)((n,d)∈Produktion ∧ n=name) ---> adaten = d --
special' = special \ {prot_id};

END get_Proddaten;

put_Proddaten(prot_id,name,adaten);

NBL ACC <=> prot_id∈special ∨ prot_id=leiter;

EFFECTS (∃(n,d)∈Produktion')(n=name ∧ d=adaten --
special' = special \ {prot_id};

END put_Proddaten;

give_special_right(prod_id,id);

PRE ACC <=> prot_id=leiter;

EFFECTS special' = special ∪ {id};

END give_special_right;

END Produktion.

Abb. 45: Abstrakter Typ 'Produktion'

werden können. Damit wird es möglich, Implementierungsprogramme des Datentyps bez. der spezifizierten Einschränkungen zu verifizieren, so daß die Einhaltung der getroffenen Vereinbarungen überprüft (von Seiten der Belegschaft) und gewährleistet (von Seiten der Firmenleitung) werden kann. Die Umgehung der vorgesehenen Einschränkungen, z.B. durch direkte Gespräche zwischen Produktionsleiter und Personalchef, wird nicht verhindert, ist aber auch nicht neu hinzugekommen.

Die Produktion erhält Zugriff auf Namen und Arbeitsdaten. Uningeschränkt abfragbar ist nur die Menge aller Arbeitsdaten. Um einzelne, namensbezogene Arbeitsdaten zu ändern, müssen sich die jeweiligen Sachbearbeiter identifizieren. Das Recht, namensbezogen zuzugreifen, wird vom Abteilungsleiter vergeben und zwar jeweils pro Zugriff. Dadurch soll die Identifikation einzelner Arbeiter erschwert werden. Die Spezifikation des Typs 'Produktion' zeigt Bild 45.

Die Schutzdaten des Datentyps enthalten die Identifikation des Produktionsleiters und eine Menge, die die Namen derjenigen enthält, die namensbezogene Zugriffe durchführen dürfen. Die Nichtverträglichkeitsrelation 'NCOMP' legt fest, daß 'put_Proddaten' mit keiner anderen Operation verträglich ist und daß 'give_special_right' nicht mit 'get_Proddaten' verträglich ist. Die Operation 'read_Proddaten' liefert alle momentan vorhandenen Arbeitsdaten, ohne die Zuordnung zu den einzelnen Arbeitnehmern. Die Ausführung der Operation ist nicht beschränkt, da keine personenbezogenen Informationen damit verbunden sind. Die Operationen 'put_Proddaten' und 'get_Proddaten' dienen zum Einfügen und Löschen von Arbeitsdaten einzelner Arbeitnehmer. Die Ausführung dieser Operationen hängt vom aufrufenden Mitarbeiter ab. Dieser muß in der Menge 'special' enthalten sein. Um das spezielle Recht, namensbezogen zuzugreifen, nur einmal pro Zuordnung benutzen zu können, wird es nach jedem Aufruf gelöscht. Zuweisen kann das Recht nur der Produktionsleiter mit der Operation 'give_special_right'. Zu bemerken ist, daß ein abstraktes Objekt 'produktionsabteilung' vom Leiter abhängt.

Bei Änderung des Produktionsleiters muß ein neues abstraktes Objekt angelegt werden. Alternativ könnte auch die Änderung des Leiters als Operation in den abstrakten Typ aufgenommen werden.

3.3.3 Das Philosophenproblem

Zunächst wird das einfache Philosophenproblem durch die Operationen 'belegen' und 'freigeben' und die zugehörigen Synchronisationseinschränkungen formuliert (Bild 46). Es wird für 'm' Philosophen wird durch einen Tisch 'phtisch' mit 'm' Gabeln charakterisiert. Die Philosophen selbst können zwei Zustände annehmen, nämlich 'denken' (schlafen) und 'essen'. Für das Essen benötigt ein Philosoph zwei nebeneinander liegende Gabeln. Der Übergang vom Denken zum Essen erfolgt über 'belegen'. Diese Operation ist nur ausführbar, wenn die zu belegende Gabel und die Gabel rechts davon frei sind. Die Operation 'freigeben' gibt zwei Gabeln wieder frei, wenn der Übergang von Essen zum Denken (Schlafen?) erfolgt.

Während der Belegungsvorgang der Gabeln eingeschränkt ist, wäre es beim Freigeben möglich, daß ein Philosoph scheinbar seine Gabeln freigibt, während in Wirklichkeit seine beiden Nachbarn essen. Damit kann er sie belegen, ohne sie tatsächlich zu besitzen, d.h., das System wird inkonsistent. Deshalb muß beim Freigeben überprüft werden, ob der Philosoph, der Gabeln freigeben will, diese besitzt. Dazu müssen die belegten Gabeln jeweils vermerkt werden. Dies wird am abstrakten Typ 'kphtisch' gezeigt (k: konsistent, Bild 47). Im abstrakten Typ 'kphtisch' wird das Philosophenproblem durch zwei Mengen 'fork' und 'eat' beschrieben. 'fork' enthält die freien Gabeln und 'eat' die essenden Philosophen. Bei der Belegung werden zwei Gabeln entfernt und der Philosoph als essend vermerkt. Eine Freigabe der Gabeln ist nur möglich, wenn der Philosoph essend ist, also die Gabeln auch belegt hat. Der Parameter 'prot_pnr' wird durch 'PROT' als geschützt gekennzeichnet. Eine mögliche Aliasbildung muß beim Aufruf der Operationen verhindert werden. Ansonsten könnte ein Philosoph die Gabeln eines anderen, nicht benachbarten Philosophen belegen.

```

ABSTRACT TYPE SPECIFICATION phtisch (m: INTEGER WHERE  $m \geq 2$ );

  ABSTRACT REPRESENTATION
    phtisch: SET OF INTEGER INIT {i |  $0 \leq i \leq m-1$ };

  PARAMETERS
    pnr: [0,..,m-1];
    a1,a2: ACTIVITY;

  OPERATIONS
    belegen(pnr);
      NBL {(pnr+1)MOD m, pnr}  $\cap$  phtisch = {};
      EFFECTS {(pnr+1)MOD m, pnr}  $\cap$  phtisch' = {};
    END belegen;

    freigeben(pnr);
      EFFECTS {(pnr+1)MOD m, pnr}  $\subset$  phtisch';
    END freigeben;

END phtisch.

```

Abb. 46: Abstrakter Typ 'phtisch'

Mit Hilfe von Rechten sollen die Belegungsmöglichkeiten der Gabeln eingeschränkt werden, um z.B. das Aushungern eines Philosophen zu verhindern. Obwohl ein typisches Synchronisationsproblem (ein Philosoph darf erst essen, wenn auch seine beiden Nachbarn gegessen haben), ist es auch mit Rechten, also als Schutzproblem, beschreibbar. Das Beispiel des Typs 'pphtisch' (p: protected, Abb.48) zeigt einerseits die enge Verwandtschaft zwischen Synchronisations- und Schutzproblemen. Andererseits enthält es auch einen Hinweis auf einen gemeinsamen Mechanismus zu ihrer Realisierung. Das Philosophenproblem wird als Menge von Sitzplätzen mit zugeordneten Gabeln ('forks') aufgefaßt. Zusätzlich wird jedem Sitzplatz eine Rechtemenge 'ph_rights' zugeordnet. Sind die Rechte 'fork_r' bzw. 'fork_l' vorhanden, so hat der dem Sitzplatz zugeordnete Philosoph das Recht, die Gabeln zu belegen. Sie werden aber erst belegt, wenn keiner der beiden Nachbarn mehr ißt. Gleichzeitig mit dem Belegen werden dem Philosophen die Gabelrechte entzogen und das Schlafrecht ('sleep', Denken im Schlaf) eingeräumt. Ein Philosoph darf die Gabeln nur freigeben, wenn er durch das Schlafrecht beweist, daß er gegessen hat. Mit dem Freigeben ordnet er seinen Nach-

```

ABSTRACT TYPE SPECIFICATION kphtisch (m: INTEGER WHERE m ≥ 2);

  ABSTRACT REPRESENTATION
    fork: SET OF INTEGER INIT {i | 0 ≤ i ≤ m-1};
    eat: SET OF INTEGER INIT {};

  PARAMETERS
    prot_pnr: [0,...,m-1];
    a1,a2: ACTIVITY;

  PROTECTION
    PROT prot_pnr;

  OPERATIONS
    belegen(prot_pnr);
      NBL {(prot_pnr+1) MOD m, prot_pnr} ∩ fork = {};
      EFFECTS
        fork' = fork \ {(prot_pnr+1) MOD m, prot_pnr} --
        eat' = eat ∪ {prot_pnr};
      END belegen;

    freigeben(prot_pnr);
      PRE prot_pnr ∈ eat;
      EFFECTS
        fork' = fork ∪ {(prot_pnr+1) MOD m, prot_pnr} --
        eat' = eat \ {prot_pnr};
      END freigeben;

END kphtisch.

```

Abb. 47: Abstrakter Typ 'kphtisch'

barn das Recht 'fork_r' (linker Nachbar) bzw. 'fork_l' (rechter Nachbar) zu, um anzuzeigen, daß er gegessen hat. Der Philosoph darf erst wieder essen, wenn seine beiden Nachbarn ihm beim Freigeben die Gabelrechte zurückgegeben haben.

ABSTRACT TYPE SPECIFICATION pphtisch (m : INTEGER WHERE $m \geq 2$);

ABSTRACT REPRESENTATION

```
forks: SET OF INTEGER INIT {i |  $0 \leq i \leq m-1$ };
ph_rights: ARRAY [0..m-1] OF rights
    INIT ( $\forall i$ ) ( $0 \leq i \leq m-1 \rightarrow$  ph_rights[i] = {fork_r, fork_l});
```

TYPES

```
rights = SET OF (fork_r, fork_l, sleep);
```

PARAMETERS

```
prot_pnr: [0, ..., m-1];
```

PROTECTION

```
PROT ph_rights, prot_pnr;
```

OPERATIONS

```
belegen(prot_pnr);
    NBL { (prot_pnr+1) MOD m, prot_pnr }  $\cap$  forks = {} --
    ACC  $\Leftarrow$  {fork_r, fork_l}  $\subset$  ph_rights[prot_pnr];
    EFFECTS
        forks' = forks \ { (prot_pnr+1) MOD m, prot_pnr } --
        ph_rights'[prot_pnr] = {sleep};
    END belegen;

freigeben(prot_pnr);
    PRE ACC  $\Leftarrow$  sleep  $\in$  ph_rights[prot_pnr];
    EFFECTS
        forks' = forks  $\cup$  { (prot_pnr+1) MOD m, prot_pnr } --
        ph_rights'[(prot_pnr+1) MOD m] =
            ph_rights[(prot_pnr+1) MOD m]  $\cup$  {fork_l} --
        ph_rights'[(prot_pnr-1) MOD m] =
            ph_rights[(prot_pnr-1) MOD m]  $\cup$  {fork_r} --
        ph_rights'[prot_pnr] = ph_rights[prot_pnr] \ {sleep};
    END freigeben;
```

END pphtisch.

Abb. 48: Abstrakter Typ 'pphtisch'

3.4 Bemerkungen

3.4.1 Vollständigkeitspostulat und "versteckter Kanal"

Die Spezifikation eines abstrakten Datentyps soll einen gegebenen oder erwünschten Sachverhalt repräsentieren. Durch die Schutz- und Synchronisationskonstrukte in SPAGAT sind neben sequentiellen auch nichtsequentielle Eigenschaften und Informationsflußeigenschaften darstellbar. Die darstellbaren Eigenschaften orientieren sich aber am Feinheitsgrad des Modells. Dies gilt besonders bei der Betrachtung von Informationsflüssen, wenn ein Element des zu modellierenden Sachverhalts nicht berücksichtigt wird. Wenn z.B. in einem Informationsflußmodell das Zugriffsverhalten des Plattenspeichers nicht repräsentiert ist, kann durch Steuerung des Zugriffsverhaltens Information fließen. Die Berücksichtigung aller sichtbaren (beobachtbaren) Eigenschaften eines Systems im zugehörigen Modell ist hinreichende Bedingung für die Kontrolle aller möglichen Informationsflüsse des Systems (Vollständigkeitspostulat, /JONE78/). Es ist offensichtlich, daß die Erfüllung dieser Voraussetzung nicht beweisbar ist. Der Hauptgrund dürfte sein, daß ein Rechensystem nicht abgeschlossen ist, sondern durch seine Schnittstellen ein offenes System bildet.

Die syntaktische Repräsentation eines abstrakten Datentyps mit SPAGAT erlaubt die Definition des Begriffs syntaktische Änderung bzw. syntaktische Abfrage. Dazu muß zunächst die Variablenmenge eines abstrakten Typs und der zugehörigen Betriebssystemmaschine unterteilt werden.

Gegeben ist ein spezifizierter abstrakter Datentyp ADT mit der Variablenmenge PAR und der Operationenmenge $\{op_1, \dots, op_n\}$. Die Komponenten der abstrakten Repräsentation sind in PAR enthalten. Dann können jeder Operation op_i zwei Parametermengen PAR_{in} und PAR_{out} folgendermaßen zugeordnet werden:

$$PAR_{in} = \{par \in PAR \mid par \text{ Eingabeparameter} \}$$
$$\cup \{komp \in PAR \mid komp \text{ kommt in 'COMP'-, 'PRIO'-, 'INF'-, 'PRE'-, 'NBL'-, 'EFFECTS'-Teil vor} \}$$
$$PAR_{out} = \{par \in PAR \mid par \text{ Ausgabeparameter} \}$$
$$\cup \{komp \in PAR \mid komp \text{ kommt in 'EFFECTS'-Teil vor} \}$$

(Ein Parameter heißt Eingabeparameter, wenn er in der Eingabeliste einer Operation enthalten ist (analog Ausgabeparameter). x PAR kommt im 'PRIO'-, 'COMP'-, 'INF'-, 'PRE'-, 'NBL'- oder 'EFFECTS'-Teil vor, wenn x in einem der sechs Teile angewandt auftritt.)

Dem spezifizierten abstrakten Typ ADT ist durch eine Interpretation I eine Betriebssystemmaschine zugeordnet. Für jede Operation $\sigma \in \Sigma$ der Betriebssystemmaschine sind zwei Variablenmengen definierbar:

$$V_{abfr}(\sigma) = \{v \in \text{Var} \mid I'(ao) \mid abfr(\sigma, v)\}$$

$$V_{aend}(\sigma) = \{v \in \text{Var} \mid I'(ao) \mid aend(\sigma, v)\}$$

V_{abfr} bezeichnet alle Variablen, die von σ semantisch abgefragt werden können, V_{aend} alle Variablen, die von σ semantisch geändert werden können. $I'(ao)$ ist die Urbildmenge des abstrakten Objekts ao vom Typ ADT. D.h., in V_{abfr} und V_{aend} werden nur die Parameter der Operationen des abstrakten Typs betrachtet.

Definition (syntaktische Abfrage, syntaktische Änderung): Gegeben sei ein abstrakter Datentyp ADT mit der Operationenmenge $\{op_1, \dots, op_n\}$. Dann heißen alle Parameter aus PAR_{in} syntaktisch abgefragt und alle Parameter aus PAR_{out} syntaktisch geändert.

Aus den Mengen PAR_{in} (PAR_{out}) eines abstrakten Datentyps bzw. aus den Mengen V_{abfr} (V_{aend}) der zugehörigen Betriebssystemmaschine läßt sich der Begriff des versteckten Kanals ableiten.

Definition (versteckter Kanal): Gegeben seien die Mengen PAR_{in} und PAR_{out} eines abstrakten Datentyps ADT, die Mengen V_{abfr} und V_{aend} der durch die Interpretation I dem Typ ADT zugeordneten Betriebssystemmaschine. Sei I^1_{PAR} die Abbildung, die jedem formalen Parameter x von ADT eine Variable $v \in \text{Var}$ zuordnet. Dann heißt

$$VK(ADT) = \{x \mid x \in V_{abfr} \mid I^1_{PAR_{in}} \vee x \in V_{aend} \mid I^1_{PAR_{out}}\}$$

die Menge der versteckten Kanäle von ADT, $x \in VK(ADT)$ heißt ein versteckter Kanal in ADT.

(x ist also versteckter Kanal, wenn x semantisch abgefragt oder geändert wird und syntaktisch nicht als Eingabe- bzw. Ausgabeparameter in Erscheinung tritt.)

Zusammenhang zwischen syntaktischer und semantischer Änderung/Abfrage: Die syntaktische Änderung/Abfrage ist keine hinreichende Voraussetzung für die semantische Änderung/Abfrage.

Begründung durch Gegenbeispiele:

1. Eine Variable aus PAR_{out} , die einen konstanten Wert in einer Operation op_i des abstrakten Typs zugewiesen bekommt, wird syntaktisch, aber nicht semantisch geändert.
2. Im folgenden Ausdruck wird der alte Zustand eines abstrakten Objekts ao syntaktisch, aber nicht semantisch abgefragt:
 $ao' = ao - ao;$

Definition (Nebenwirkungsfreiheit): Ein spezifizierter abstrakter Datentyp ADT ist frei von versteckten Kanälen, wenn gilt:

$$VK(ADT) = \emptyset$$

Satz: Ein abstrakter Datentyp ADT ist genau dann nebenwirkungsfrei, wenn die syntaktische Änderung und Abfrage notwendige Voraussetzung der semantischen Änderung und Abfrage ist.

Beweis:

Syntaktische Änderung/Abfrage ist notwendige Voraussetzung der semantischen Änderung/Abfrage

$$<---> V_{abfr} \subset I' | PAR_{in} \wedge V_{aend} \subset I' | PAR_{out}$$

$$<---> VK(ADT) = \emptyset$$

Zur Untersuchung von Informationsflüssen ist die Nebenwirkungsfreiheit zu fordern. Dadurch wird es möglich, durch statische Untersuchungen eines spezifizierten abstrakten Datentyps mögliche Informationsflüsse zu erkennen. Mit "statischen Untersuchungen" ist die Betrachtung der Mengen

PAR_{in} und PAR_{out}

gemeint. Aus der Nebenwirkungsfreiheit folgt, daß

$$V_{abfr} \subset I' | PAR_{in} \wedge V_{aend} \subset I' | PAR_{out}$$

$$---> (I' | PAR_{in} = \emptyset \quad ---> V_{abfr} = \emptyset$$

$$\wedge \quad I' | PAR_{out} = \emptyset \quad ---> V_{aend} = \emptyset)$$

Aus den Mengen PAR_{in} und PAR_{out} können also die nicht möglichen Informationsflüsse eines abstrakten Typs ADT bestimmt werden. Bei der Bestimmung von Verifikationsbedingungen zur konsistenten Spezifikation und korrekten Implementation asynchroner, geschützter abstrakter Datentypen wird dieser Gesichtspunkt noch genauer erläutert (Kapitel 5).

3.4.2 Einordnung

In Kapitel 1 wurde bereits gezeigt, daß Sicherheit (sicher(1), Kap. 1.1) ein relativer Begriff ist. Die vorgeschlagene Spezifikationsmethode erhöht durch zwei Eigenschaften den Grad der Sicherheit eines abstrakten Typs. Einerseits erfordert die formale und problemnahe Vorgehensweise exakte Vorstellungen des zu spezifizierenden Datentyps. Andererseits enthält SPAGAT spezielle Konstrukte zur Formulierung von Synchronisations- und Informationsflußeinschränkungen. Dadurch sind die in einem abstrakten Datentyp enthaltenen Synchronisations- und Schutzprobleme explizit darstellbar und formal handhabbar.

Das 'INF'-Konstrukt führt zu einer echten Erweiterung der Methode, da in den Konstrukten 'PRE' und 'NBL' nur Ausführungseinschränkungen einzelner Operationen beschreibbar sind. Der 'SYN'-Teil betrachtet dagegen nicht die semantischen Änderungen oder Abfragen einer Aktivität, sondern nur ihre Beziehungen zu anderen Aktivitäten. Im 'INF'-Konstrukt werden zusätzlich Beziehungen zwischen Operationsklassen, den Operationen eines abstrakten Typs, unabhängig von konkreten Aktivitäten betrachtet.

In Kapitel 1.1 werden drei Benutzerklassen einer Rechenanlage definiert, denen die sich teilweise widersprechenden Schutzanforderungen Privatheit, Sicherheit (sicher(2)) und Integrität zugeordnet sind. Im weiteren soll diskutiert werden, inwieweit diese Forderungen durch die angegebenen Schutzkonstrukte erfüllt werden können.

Privatheitsforderungen müssen in die formale Spezifikation von Programmen übernommen werden, die Erfüllung der Forderungen muß nachvollziehbar und beweisbar sein. Die Problembezogenheit der Methode erleichtert die Übertragung der Anforderungen auf das Modell. Auch zur Formulierung von Schutzeinschränkungen sind die repräsentierenden Datenstrukturen frei wählbar. Daraus ergibt sich auch die Verständlichkeit der formulierten Bedingungen. Die Beweisbarkeit der Schutzbedingungen folgt aus der formalen Vorgehensweise.

Integritätsanforderungen, die der Betreiber einer Rechenanlage stellt, führen zu möglichst einfachen, ausdrucksstarken Schutzkonstrukten, die an andere Software-Entwicklungshilfsmittel angepaßt sind. Dadurch reduziert sich der Aufwand zur Anpassung an verschiedene Schutzprobleme. Die Einbettung in vollständige Software-Entwicklungsmethoden vereinfacht den Aufwand zur Integration von Schutzproblemen. Da SPAGAT nicht speziell zur Formulierung von Schutzproblemen dient und in eine Methode zur Spezifikation und korrekten Implementation abstrakter Datentypen eingebettet ist, sind obige Forderungen zum Teil erfüllt. Die Ausdrucksstärke ergibt sich aus der Definition der Konstrukte mit Hilfe der Prädikate `abfr` und `aend` der erweiterten Betriebssystemmaschine. Trotz der Allgemeinheit der Konstrukte lassen sich Konsistenzbedingungen für die Spezifikation angeben (siehe Kapitel 5).

Sicherheitsanforderungen unterscheiden sich nur hinsichtlich der Allgemeinheit der Konstrukte, da im Benutzerbereich die Vielfalt von Schutzproblemen in verschiedenen Software-Systemen zu berücksichtigen ist.

4 Schutzkonstrukte in Programmiersprachen

Obwohl Basisschutzmaßnahmen in Programmiersprachen seit langem existieren (Blockkonzept, Klassenkonzept), gibt es kaum darüber hinausgehende Ansätze (siehe auch Abschnitt 1.2.3). Einer der wesentlichen Gründe dafür dürfte die hohe Maschinenabhängigkeit von Synchronisations- und Schutzmaßnahmen sein. Ein zweiter Grund ist vermutlich die Realisierung größerer Programme in Form einzelner, unabhängiger Prozesse. Das bedeutet, daß die Datenbereiche eines Programms nur für einen Prozeß (als einzelne aktive Einheit) sichtbar sind. Prozeßinteraktionen bleiben dem Programmierer selbst überlassen, indem z.B. Routinen des Betriebssystems von den Programmen aus ansprechbar sind. Auch die Bearbeitung von Objekten, die von der Lebensdauer einzelner Prozesse unabhängig sind, ist in Benutzerprogrammen nur über spezielle Schnittstellen, z.B. in Datenbanksystemen, möglich. Daraus entsteht auch die Diskrepanz zwischen Systemimplementierungssprachen und Anwendungsprogrammiersprachen. Bei der Systemprogrammierung können Synchronisationsaspekte nicht vernachlässigt werden und die Lebensdauer von Objekten ist i.allg. unabhängig von den einzelnen Prozessen. Dabei macht es keinen Unterschied, ob das Betriebssystem auf einem Prozeß- oder einem Prozedurkonzept beruht. Schutzkonzepte bleiben, bis auf die Trennung einzelner Programmteile (Modulkonzept) auch in Systemimplementierungssprachen unberücksichtigt, da Schutzmaßnahmen erst zu implementieren sind, also nicht zur Implementierung verwendet werden können. Dieser Aspekt ist bei der Verwendung von Sicherheitskernen und Hardware-Mechanismen nicht mehr gültig. Während zur Implementation des Sicherheitskerns eines Systems Schutzmaßnahmen der Hardware verwendet werden, können alle weiteren Systemteile auf dem Sicherheitskern aufbauen. Gerade in Systemimplementierungssprachen könnte die Verwendung von Schutzkonstrukten, für die auch formale Verifikationskriterien vorliegen, die Zuverlässigkeit und damit die Sicherheit erhöhen, eine wichtige Voraussetzung für die Sicherheit und Zuverlässigkeit darauf aufbauender Software-Systeme.

Auch in der Anwendungsprogrammierung könnten Schutzkonstrukte den Korrektheitsbeweis von Schutzaussagen und ihre Implementa-

tion vereinfachen. Insbesondere bei benutzerdefinierten abstrakten Objekten (Objekte von abstrakten Datentypen, die auf Benutzerebene definiert sind), die mehreren Benutzern und Benutzerprozessen gemeinsam sind (analog zum bisherigen Typ Dateiverwaltung), müssen Schutzaspekte auf Problemniveau berücksichtigt werden. Die Programmierung mit Hilfe abstrakter Datentypen, deren Abstraktionskonzepte zum Teil vom Betriebssystem zur Verfügung gestellt werden (HYDRA, PSOS), unterscheidet sich damit von der monolithischen Programmierung, wo die Interaktion zwischen Prozessen auf wenige vordefinierte Kanäle beschränkt ist. Durch die Interaktion über abstrakte Objekte wird die Trennung der Datenbereiche verschiedener Programme und Prozesse aufgehoben, die in monolithischen Programmen vor gegenseitigen Beeinflussungen geschützt hat. Trotz des Aufbrechens der Programme wären gegenseitige Zugriffsbeschränkungen beim Übersetzen kontrollierbar, wenn die Objekte zur Übersetzungszeit bekannt wären und jedem Prozeß eine feste Art von Zugriffen zugeordnet wäre. Da dies i.allg. nicht erfüllt ist, müssen zur Lösung von Schutzproblemen dynamische Kontrollen durchgeführt werden, die auch vom Programmierer steuerbar sind. Ähnliche Probleme ergaben sich bei der Einführung von Timesharing-Systemen, da der natürliche Schutz des Einbenutzerbetriebs entfiel.

Bild 49 zeigt die Einordnung abstrakter Datentypen an der Schnittstelle zwischen Betriebssystem und Anwendungsprogrammen, unter Berücksichtigung eines Sicherheitskerns im Betriebssystem. Analog könnte statt des Sicherheitskerns ein Synchronisationskern eingesetzt werden, der einen grundlegenden Synchronisationsmechanismus repräsentiert, auf dem alle weiteren Systemteile aufbauen.

In den folgenden Abschnitten werden zunächst allgemeine Schutzkonstrukte für eine Programmiersprache vorgestellt. Als Rahmen wird dazu eine Modula-2-Erweiterung (/WIRT80/, /RUPP83/) verwendet. Die Erweiterung enthält Synchronisationskonstrukte, die sich aus den entsprechenden SPAGAT-Konstrukten ableiten und erlaubt die Implementation asynchroner abstrakter Datentypen. Auch die Schutzkonstrukte sollen in der Syntax weitestgehend den SPAGAT-Konstrukten entsprechen, um durch eine einheitliche

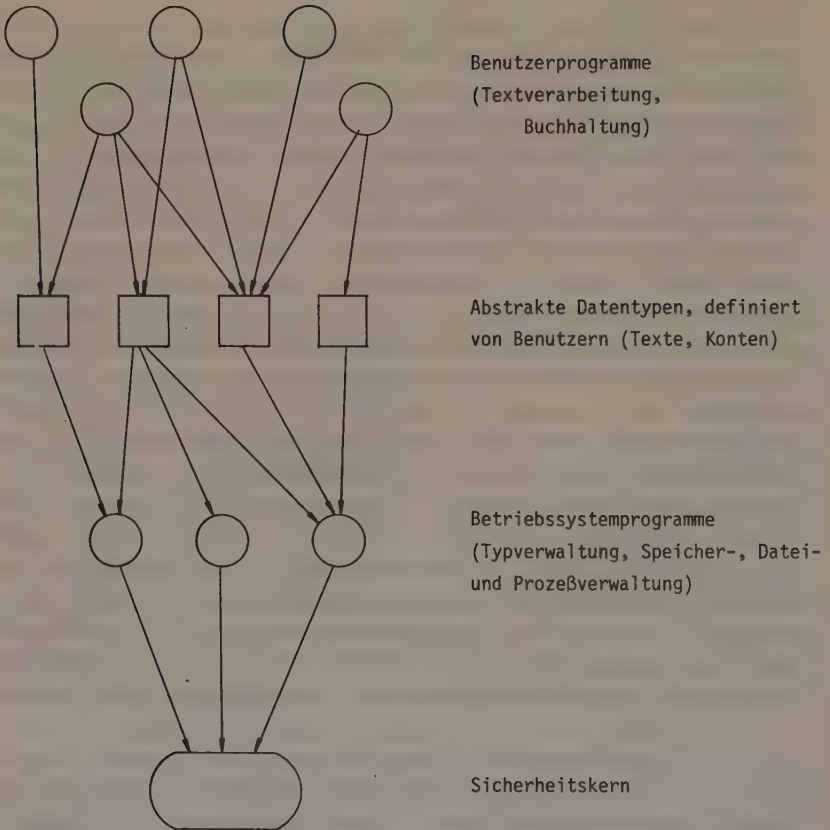


Abb. 49: Abstrakte Typen an der Schnittstelle zwischen Betriebssystem und Anwendungsprogramm

Syntax in Spezifikation und Implementation die Zusammenhänge transparent zu machen. Der dritte Abschnitt enthält Vorschläge zur Übersetzung der Konstrukte und ihre Auswirkungen auf das Laufzeitsystem. Der vierte Abschnitt soll anhand zweier Beispiele die Wirkung der Schutzkonstrukte verdeutlichen. Der fünfte Abschnitt beschreibt eine Implementierungsmöglichkeit auf dem System VAX 11/780 unter dem Betriebssystem VMS, Version 3.1.

4.1 Grundlagen zur Implementation abstrakter Datentypen

4.1.1 Implementation und angewandtes Auftreten abstrakter Datentypen

Um eine Sprache zur Implementation abstrakter Datentypen erweitern zu können, muß geklärt werden, was die Implementation eines abstrakten Typs ist. Mit Hilfe abstrakter Datentypen soll die Implementation von Software-Systemen vereinfacht werden, indem die Deklaration von Objekten auf Problemniveau möglich wird. Zusätzlich wird die Bearbeitung der Objekte vereinfacht, da die Typoperationen als komplexe Operatoren auf Objekten des abstrakten Typs angesehen werden können. Es muß also zwischen dem abstrakten Typ und den Objekten dieses Typs unterschieden werden. In Kapitel 3 wird bereits auf die Unterscheidung zwischen der Spezifikation von Software-Systemen und der Spezifikation abstrakter Typen hingewiesen. Dies wird auch bei der Implementation beibehalten.

Da im weiteren die Implementation von abstrakten Datentypen betrachtet wird, soll informell ein Bezug zwischen abstrakten Objekten und abstrakten Datentypen hergestellt werden. Bei der Deklaration eines abstrakten Objekts sind die Datenbereiche, die den Objektzustand repräsentieren, anzulegen. Enthält die Beschreibung des Objekttyps generische Parameter, d.h., ist bei der Übersetzung des abstrakten Typs die Repräsentation nur zum Teil festgelegt, muß die Deklaration die generischen Parameter so spezifizieren, daß die Repräsentation bestimmt werden kann. Zusätzlich muß der abstrakte Datentyp selbst, der im wesentlichen aus den Typoperationen besteht, bereitgestellt werden. Bei der Implementation von Synchronisations- und Schutzproblemen muß der abstrakte Typ um Operationen und Daten zur Realisierung der programmierten Synchronisations- und Schutzeinschränkungen erweitert werden. Während die Schutzeinschränkungen eines abstrakten Typs die Zugriffe regeln, die über die Typoperationen ausführbar sind, ist das umgebende Software-System eines abstrakten Objekts für die Weitergabe des Objekts an Prozesse und für die korrekte Versorgung der Schutzparameter verantwortlich. Sind die Schutzeinschränkungen eines abstrakten Typs unabhängig

von Parametern, so hat das umgebende Software-System keinen Einfluß auf den Schutz der abstrakten Objekte. Schließlich sind abstrakte Typen selbst in den Programmen sichtbar zu machen, die zugehörige abstrakte Objekte deklarieren. Dazu stehen die bekannten "IMPORT"-Mechanismen aus ADA oder Modula-2 zur Verfügung. Der Import ist dabei auf die Typoperationen und den Typnamen beschränkt.

Abstrakte Datentypen sollen nicht nur innerhalb von Programmen die problemangepaßte Implementation unterstützen, sondern auch die Kommunikation zwischen verschiedenen Prozessen vereinfachen. Dazu ist der Existenzbereich von abstrakten Objekten vom Existenzbereich der deklarierenden Prozesse zu trennen, wie es bereits beim Datentyp Datei der Fall ist.

4.1.2 Die Sprachumgebung

In /RUPP83/ wird eine Sprache zur Implementation asynchroner abstrakter Datentypen vorgestellt, deren Kontrollstrukturen und Typdeklarationen aus Modula-2 stammen (/WIRT80/). Diese Sprache wird in Abschnitt 2 um Schutzkonstrukte erweitert. Die wesentlichen Sprachelemente werden am bereits spezifizierten Beispiel des Typs 'nezev' eingeführt (Bild 50).

Durch 'ABSTRACT TYPE IMPLEMENTATION nezev

(Length: CARDINAL; elemtyp: TYPE)'

wird die Implementation des abstrakten Typs 'nezev' eingeleitet. Der abstrakte Typ ist parametrisiert, d.h., die Operationen und der notwendige Speicherplatz sind erst bestimmbar, wenn in einer Objektdекlaration der Parameter 'Length' und der Typ 'elemtyp' konkretisiert werden. Im 'TYPE'-Teil sind zusammengesetzte Arten, die in der Implementation verwendet werden, definierbar. Falls Synchronisationsbedingungen angegeben werden, muß der 'TYPE'-Teil die dort benötigten Synchronisationskontrollblöcke ('PCB') enthalten. Darin werden als Aufzählungstyp die Operationsnamen der in den Synchronisationsbedingungen vorkommenden Operationen definiert (standardmäßig unter der

```

ABSTRACT TYPE IMPLEMENTATION nezev
  (Length: CARDINAL; elemtyp: TYPE);

TYPE PCB = RECORD
  operation: (insert,remove);
  prio: CARDINAL;
END;

CONCRETE REPRESENTATION
  elements: ARRAY [0..Length-1] OF elemtyp;
  inelem, outelem: CARDINAL INIT inelem:= 0; outelem:= 0 END;

SYN
  COMP(pc1,pc2: PCB) <=> (pc1.operation=insert) AND
    (pc2.operation=remove) AND
    (inelem-outelem>1);
  PRIO(pc1,pc2: PCB) <=> (pc1.operation=insert) AND
    (pc2.operation=remove) AND
    (pc1.pr>pc2.pr);

OPERATIONS

PROCEDURE insert(pr: CARDINAL; elem: elemtyp);
  NBL inelem-outelem<Length;
  BEGIN
    elements[inelem MOD Length]:= elem;
    INC(inelem);
  END insert;

PROCEDURE remove(VAR elem: elemtyp);
  NBL inelem>outelem;
  BEGIN
    elem:= elements[outelem MOD Length];
    INC(outelem);
  END remove;

END nezev.

```

Abb. 50: Implementation des Typs 'nezev'

Komponente 'operation':

'operation: (append, remove);'

Konflikte mit den Operationen selbst treten nicht auf, da diese ausschließlich im Operationsteil vorkommen. Weitere Komponenten eines Synchronisationskontrollblocks sind zur Synchronisation benötigte Parameter: 'pr: CARDINAL;'. Im Teil 'CONCRETE REPRESENTATION' wird die konkrete Repräsentation eines abstrakten Objekts mit Hilfe von Standardtypen und von benutzerdefinierten Typen definiert. Mit 'INIT' werden Initialisierungsbedingungen der abstrakten Objekte des Typs vorgegeben. Im Teil 'VAR' werden Hilfsvariablen zur Formulierung der sequentiellen Abläufe

deklariert. Dafür entfällt der Teil 'PARAMETERS' der Spezifikation, da die Operationsparameter wie in Modula-2 im Prozedurkopf spezifiziert werden. Zusätzlich können in einem Konstantenteil ('CONST') konstante Größen eines abstrakten Typs eingeführt werden. Elemente anderer abstrakter Typen sind mit der 'IMPORT'-Klausel zu importieren ('CONST' und 'VAR' sind im Beispiel nicht enthalten).

Der 'SYN'-Teil legt mit Hilfe eines in 'TYPE' definierten Datentyps die Verträglichkeits- und Prioritätsrelation des abstrakten Typs fest. Einschränkend gegenüber der Spezifikation müssen die Bedingungen der Modula-2-Syntax für logische Ausdrücke genügen. Der aus der Bedingung zu berechnende Wert ist als Wert einer Funktionsprozedur 'COMP' bzw. 'PRIO' anzusehen. Die Parameter in 'COMP' und 'PRIO' ('pcl', 'pc2') benennen die in den Synchronisationsbedingungen verwendeten Synchronisationskontrollblöcke.

Im 'OPERATIONS'-Teil schließlich werden die sichtbaren Prozeduren des Datentyps angegeben. Falls Hilfsprozeduren, die nicht zu den sichtbaren Operationen des Datentyps gehören, programmiert werden sollen, steht ein spezieller Teil 'DEFINITIONS' (entfällt im Beispiel) zur Verfügung. Die dort implementierten (Funktions-)Prozeduren sind nicht exportierbar oder anderweitig importierbar und können auch nicht als zu synchronisierende Operationen verwendet werden. Sie gehören also nicht zur Aktivitätsmenge. Während der Prozedurkörper der Modula-2-Syntax folgt, kann im Teil 'NBL' die Ausführung einer Operation, in Abhängigkeit vom Wert der zugehörigen Bedingung, blockiert werden.

4.1.3 Semantik

Analog zur Spezifikationssprache kann auch ein implementierter abstrakter Datentyp als Interpretation einer Betriebssystemmaschine aufgefaßt werden. Der sequentielle Teil der Operationen wird durch die Semantik der verwendeten Programmiersprache bestimmt, wobei die Operationen als Prozeduren aufzufassen sind und die Variablenmenge Var auf die spezifizierten Parameter abgebildet wird.

Der nichtsequentielle Teil eines implementierten abstrakten Typs wird über die Synchronisationskonstrukte ('COMP', 'PRIO') auf die Betriebssystemmaschine abgebildet. Die aufgerufenen Prozeduren eines abstrakten Typs bilden die Aktivitätsmenge, die übergebenen Parameter sind in den Synchronisationsbedingungen verwendbar. Die Semantik der Bedingungen wird von der Bedeutung der gleichnamigen Prädikate der Betriebssystemmaschine bestimmt.

4.2 Schutzkonstrukte

Ausgehend von den in Kapitel 3 vorgestellten Schutzkonstrukten sind drei wesentliche Komponenten zu berücksichtigen. Einerseits werden für Schutzprobleme spezielle Datenstrukturen benötigt, um die gewünschten Eigenschaften zu implementieren. Da Schutzeinschränkungen auch von den aufrufenden Prozessen abhängen, muß eine Schnittstelle zur Umgebung des jeweiligen Datentyps geschaffen werden. Dazu werden Schutzobjekte zur Verfügung gestellt. Andererseits sollen einzelne Operationsaufrufe eingeschränkt werden können (Zugriffsschutz). Dazu steht in der Spezifikationssprache das 'ACC'-Konstrukt zur Verfügung, das auch in die Programmiersprache übernommen werden soll. Dabei ist zunächst der 'PRE'-Teil einer Operation in die Implementationsprache zu integrieren. Schließlich bleibt die Informationsflußkontrolle eines abstrakten Datentyps. Das allgemeine

'INF'-Konstrukt der Spezifikation kann nicht übernommen werden, sondern muß geeignet eingeschränkt werden. Wesentlich für die Konstrukte ist, daß keine speziellen Mechanismen sprachlich repräsentiert werden, sondern Schutz einschränkungen in Abhängigkeit vom zu implementierenden Problem formuliert werden können. Die Beschreibung der Syntax und der statischen Semantik verwendet dieselbe Notation wie in Kapitel 3. Auch die Bedeutung der Attribute wird übernommen. Das zusätzliche Attribut 'anzahl' zählt die Anzahl der 'PRE'-Bedingungen einer Operation. Im folgenden wird nur die Syntax der Schutzkonstrukte vorgestellt. Die vollständige Syntax der erweiterten Sprache wird nicht angegeben. Die zugrunde liegende Sprache wird in /RUPP83/ beschrieben. Die erweiterte Sprache soll im folgenden IMAGAD (Implementierungssprache für asynchrone geschützte abstrakte Datentypen) genannt werden.

4.2.1 Schutzobjekte

Das 'PROT'-Konstrukt dient in der Spezifikation zur Kennzeichnung einer Komponente der abstrakten Repräsentation als Schutzkomponente. Analog kann in der konkreten Repräsentation eine Kennzeichnung durchgeführt werden (siehe Abb.51). Der Index 'i' in runden Klammern soll die Wiederholungen des nicht-terminalen Symbols kennzeichnen. Dabei wird eine Komponente der Repräsentation des abstrakten Typs als Schutzobjekt gekennzeichnet, so daß darauf in einer 'INF'- oder 'ACC'-Bedingung zugegriffen werden kann. Die Vereinbarung einer Schutzkomponente ist in die konkrete Repräsentation integriert. Damit sind Schutzkomponenten auch initialisierbar.

Parameter zur Steuerung von Schutzüberprüfungen müssen auch in der Implementation von anderen Parametern getrennt werden, um ihre willkürliche Manipulation in den aufrufenden Programmen zu unterbinden. Die Kontrolle der Parameter außerhalb des Datentyps kann nicht innerhalb der Typimplementation erfolgen. Aber die Parameterübergabe kann speziell eingeschränkt werden. Dazu wird ebenfalls das Attribut 'PROT' verwendet, das die be-

```

schutzteil = [PROTECTION {schutzobjekte ","}
               {informationsfluss ","} ].

schutzobjekte = PROT identifikator { "," identifikator } .
    identifikator.prot:= TRUE;

praebedingungen = [ PRE ausf_bedingungen ]
                   [ NBL { ausf_bedingung "--" } ].

ausf_bedingungen = { ausf_bedingung(i) "--"}.
    ausf_bedingungen.anzahl:= max(i|ausf_bedingung(i).wert#NIL);

ausf_bedingung = bedingung(1) <1> |
                ACC "<=>" bedingung(2) <2> .
    <1> ausf_bedingung.wert:= bedingung(1).wert;
    <2> ausf_bedingung.wert:= bedingung(2).wert;
    <2> (∃v∈bedingung(2).var)(v.prot ---> FEHLER);

informationsfluss = inf_funktion "<=>" bedingung .
    (∃v∈bedingung.var)(¬v.prot ∧ v.typ#ACTIVITY) ∨
    (v.typ=ACTIVITY ∧ ¬v∈inf_funktion.var) ---> FEHLER);

inf_funktion = (INF|NINF)[identifikator(1) "," identifikator(2)].
    inf_funktion.var:= identifikator(1).var U identifikator(2).var;
    (∃v∈inf_funktion.var)(v.typ#ACTIVITY ---> FEHLER);

```

Abb. 51: Die Schutzkonstrukte der Implementationssprache

reits vorhandenen Parameterübergabemechanismen dahingehend erweitert, daß die mit 'PROT' attribuierten Parameter in der Implementation des abstrakten Typs nur eingeschränkt verwendet werden dürfen (Abb.51). Die Einschränkungen sind analog zur Spezifikation und werden bei den Informationsfluß- und Zugriffseinschränkungen noch deutlich. Die Kennzeichnung von Schutzparametern dient

- einerseits zur Einschränkung der in Informationsfluß- und Zugriffsbedingungen vorkommenden Variablen,
- andererseits zur syntaktischen Kennzeichnung der für Schutzprädikate notwendigen Parameter.

Schutzparameter und Schutzkomponenten eines abstrakten Typs bilden zusammen die Schutzobjekte, die zu Informationsfluß- und Zugriffseinschränkungen verwendet werden können.

Änderung von Schutzobjekten

Durch die Einbettung von Schutzobjekten in die Repräsentation bzw. die Kennzeichnung von Parametern als Schutzobjekte sind Schutzobjekte in den Operationen abfragbar und änderbar. Die Konsistenz der Änderungen kann durch Berücksichtigung der Schutzobjekte in den Synchronisationsbedingungen eingehalten werden. Die Gleichbehandlung von Schutzobjekten und den abstrakten Objekten bzw. den Parametern erhöht die Übersicht und Verständlichkeit. Auch die Änderung von Schutzobjekten kann durch weitere Schutzobjekte kontrolliert werden. Die dabei entstehende Schachtelung von Schutzobjekten muß durch Einführung eines konstanten Schutzobjekts abgebrochen werden (vgl. Rekursion). Ein Beispiel dafür ist der Datentyp 'Produktion' in Abschnitt 3.3. Der Datentyp ist parametrisiert bez. des Schutzobjekts 'distributor', das als generischer Parameter bei der Erzeugung eines abstrakten Objekts des Typs vorzugeben ist. Soll ein anderer Identifikator für 'distributor' eingesetzt werden, ist ein neues Objekt zu erzeugen.

4.2.2 Zugriffseinschränkungen

Zur Formulierung von Zugriffseinschränkungen ist einerseits ein 'PRE'-Konstrukt zur Formulierung von Nichtausführbarkeitsbedingungen einer Operation einzuführen. Daraus ergeben sich Probleme beim Operationsabbruch. Andererseits ist das 'ACC'-Konstrukt zur Formulierung von Zugriffseinschränkungen in die möglichen 'NBL'- und 'PRE'-Bedingungen aufzunehmen.

Das 'PRE'-Konstrukt ist syntaktisch analog zum 'NBL'-Konstrukt, wie aus Bild 51 ersichtlich. Ist eine 'PRE'-Bedingung bei Aufruf einer Operation nicht erfüllt, wird die Ausführung abgebrochen. Mit dem Abbruch des Aufrufs ist eine Fehlermeldung zu verbinden, um zwischen mehreren 'PRE'-Bedingungen unterscheiden zu können. Jeweils eine mit "--" abgeschlossene Bedingung wird als 'PRE'-Bedingung bezeichnet und bekommt eine Fehlermeldung zugeordnet. Um den Mechanismus zur Fehlermeldung

zu vereinfachen, wird eine Fehlernummer automatisch aus der Bedingungsfolge generiert. Die Nummer entspricht dem Index der Bedingung in der Folge. Der Erfüllung aller Bedingungen ist die Nummer 0 zugeordnet. Deshalb wird das Attribut 'anzahl' eingeführt, das die maximale Zahl der 'PRE'-Bedingungen einer Operation enthält.

Das 'ACC'-Prädikat läßt sich damit einfach in den vorgegebenen Rahmen einbetten (siehe Abb.51). Das nichtterminale Symbol 'ausf-bedingung' wird auch im 'NBL'-Konstrukt verwendet, so daß das 'ACC'-Konstrukt sowohl zur Blockierung einer Operation als auch zu ihrem Abbruch führen kann. Das nichtterminale Symbol 'bedingung' entspricht Modula-2-Bedingungen. Es wird zur Definition des 'INF'-Konstrukts ebenfalls verwendet.

4.2.3 Informationsflußeinschränkungen

Da sich Informationsflußeinschränkungen nicht auf einzelne Operationen beziehen, sondern auf Operationspaare, werden sie zentral im Teil 'PROTECTION' zusammengefaßt, wo auch die Schutzattribute eines Typs definiert werden (siehe Bild 51). In der Informationsflußbedingung können die Aktivitätsparameter (des 'INF'-Konstrukts) und Schutzobjekte vorkommen. Als Aktivität kann auch die konstante Aktivität 'INIT_ACTIVITY' angegeben werden. Die Semantik des 'INF'-Konstrukts muß gegenüber der Spezifikation eingeschränkt werden, da in einer Implementation der Mechanismus zur dynamischen Realisierung der 'INF'-Bedingungen die Bedingungen selbst verletzen kann. Darin, daß eine Operation wegen einer verletzten 'INF'-Bedingung abgebrochen wird, kann selbst Informationsfluß enthalten sein. Die Alternative, die 'INF'-Bedingungen durch statische Überprüfungen zu garantieren, würde zu einer Einschränkung der möglichen Informationsflüsse führen, da mit "Worst-case"-Annahmen zu argumentieren wäre. Auch in diesem Fall folgt aus einer nichterfüllten 'INF'-Bedingung nur das nichterfüllte inf-Prädikat der Spezifikation, nicht umgekehrt.

Trotz der möglichen Informationsflüsse wird ein dynamischer Mechanismus zur Informationsflußprüfung vorgeschlagen, der nur noch den versteckten Kanal der Informationsflußbedingung selbst enthält. Es wird angenommen, daß die Übertragungsrate über den versteckten Kanal so gering ist, daß die Übertragung relevanter Information äußerst aufwendig und damit unwahrscheinlich wird. Die Verwendung des versteckten Kanals kann aber erkannt und in einem Logbuch vermerkt werden, so daß eine Verfolgung im Nachhinein möglich wird. Soll auch dieser Kanal unterbunden werden, müssen statische Analysen durchgeführt werden. Entsprechende Maßnahmen zeigt Kapitel 5.

Die Semantik der Informationsflußeinschränkung in der Implementation eines abstrakten Typs kann durch das inf -Prädikat der zugehörigen Betriebssystemmaschine definiert werden. Sei

$\text{IADT} = (\text{OP}, \text{AOPAR}, \text{INF}, \text{COMP}, \text{PRIO})$

die Interpretation der erweiterten Betriebssystemmaschine

$\text{bsm} = (\bar{P}, D, \text{Var}, R, \text{inf}, \text{comp}, \text{prio})$.

AOPAR enthält die Interpretation der Variablenmenge Var. Während AO die Komponenten des Datentyps bezeichnet, enthält PAR die Operationsparameter. Zusätzlich werden für jedes abstrakte Objekt drei weitere Parameter implizit eingeführt, die die versteckten Kanäle über ein Objekt eines abstrakten Typs repräsentieren. Mit 'NBL PAR' wird ein Ausgabeparameter bezeichnet, der Informationsflüsse über die 'NBL'-Bedingung repräsentiert. Diese können nicht ausgeschlossen werden, wenn die Abarbeitungsdauer einer Operation für einen aufrufenden Prozeß beobachtbar ist. 'PRE PAR' bezeichnet einen Ausgabeparameter, der die Fehlermeldungen repräsentiert, die über die 'PRE'-Bedingungen der Operationen beim Abbruch entstehen. Schließlich bezeichnet 'INF PAR' den Informationsfluß, der durch Operationsabbruch wegen Verletzung der 'INF'-Bedingung entsteht. Es wird angenommen, daß die drei Parameter Ausgabeveriablen jeder Operation sind und damit semantisch abgefragt werden können. Semantische Änderung kann bei jedem Operationsaufruf erfolgen, der die in den Bedingungen überprüften Zustände des abstrakten Objekts ändert.

Mit der Komponentenmenge AO des Typs IADT kann der Informationsfluß in IADT wie folgt angegeben werden:

$$\begin{aligned} \text{inf}(\sigma_1, \sigma_2, o, B_1, B_2) &\leftarrow \text{INF}(a_1, a_2) \wedge a_1 = I(\sigma_1) \wedge a_2 = I(\sigma_2) \\ &\wedge (I(o) = \text{ao} \vee I(o) = \text{NBL_PAR} \vee \\ &I(o) = \text{PRE_PAR} \vee I(o) = \text{INF_PAR}) \end{aligned}$$

(Informationsfluß über das abstrakte Objekt ist also höchstens dann möglich, wenn die 'INF'-Bedingung dem inf-Prädikat der zugehörigen Betriebssystemmaschine entspricht oder wenn die versteckten Kanäle des Datentyps verwendet werden. Die Äquivalenz wird nicht verlangt, da die zur dynamischen Informationsflußuntersuchung notwendigen Variablenmengen statisch berechnet werden (siehe Abschnitt 4.3). Die Bedingungen B_1 und B_2 sind in der 'INF'-Bedingung enthalten.)

4.3 Übersetzung der vorgestellten Konstrukte und ihre Laufzeitunterstützung

Die Erzeugung abstrakter Objekte

In Abschnitt 4.1 wurde bereits auf die Diskrepanz zwischen der Implementation abstrakter Datentypen und der Erzeugung von abstrakten Objekten eingegangen. Ein abstrakter Datentyp kann in Ada in Form eines 'PACKAGE' implementiert werden, das die Operationen des Typs und den Typnamen mit der Einschränkung 'LIMITED PRIVATE' exportiert. In anderen Programmteilen können dann Objekte dieses Typs deklariert werden. Der Übersetzer kann Zugriffe auf Objekte des Typs überprüfen und auf die definierten Typoperationen beschränken. Abstrakte Typen können in Ada auch parametrisiert ('GENERIC') sein.

Wesentlich am vorliegenden Typkonzept ist die gemeinsame Verwendung eines abstrakten Objekts durch mehrere Prozesse, insbesondere Prozesse verschiedener Benutzer. Diese Eigenschaft läßt

sich nicht allein mit dem 'TASK'-Konzept aus Ada realisieren, da Prozesse verschiedener Benutzer zur Kommunikation auf das Prozeßkonzept des zugrunde liegenden Betriebssystems zugreifen. Der Benutzerbegriff ist deshalb notwendig, da die Zugriffsmöglichkeiten eines Benutzers auf die von ihm erzeugten Objekte meist konstant und damit bereits bei der Übersetzung kontrollierbar sind. In der vorliegenden Arbeit geht es aber um die dynamische Zugriffs- und Informationsflußkontrolle.

Die Unterscheidung zwischen dem Zugriff eines oder mehrerer Prozesse auf ein Objekt hat Auswirkungen auf die Semantik des 'PRE'- und des 'NBL'-Konstrukts einer Operation. Wird ein Prozeß wegen einer 'NBL'-Bedingung blockiert und kann kein anderer Prozeß den Objektzustand (und damit den Wert der Blockierungsbedingung) ändern, so wäre der Prozeß unendlich lange blockiert. Wenn man bei der Deklaration der Objekte Angaben über die Anzahl der zugreifenden Prozesse erlaubt, kann automatisch die 'NBL'-Bedingung als 'PRE'-Bedingung interpretiert werden, so daß keine Blockierung, sondern der Abbruch der aufgerufenen Operation erfolgt. Damit kann die Verklemmungsgefahr nicht ausgeschlossen, aber zumindest eingeschränkt werden.

4.3.1 Schutzobjekte

Zur Implementation der Synchronisationsbedingungen wird in /RUPP83/ der Datentyp 'SynPCB' (Synchronisationskontrollblock) verwendet, ein Strukturtyp, der bei der Implementierung zu deklarieren ist. Ein Objekt dieses Typs wird bei jedem Operationsaufruf angelegt. Die erste Komponente ist von einem Aufzählungstyp, der als Elemente alle sichtbaren Operationsnamen des implementierten abstrakten Typs enthält. Die folgenden Komponenten enthalten die Werte von Operationsparametern, die in einer 'PRIO'- oder 'COMP'-Bedingung vorkommen, die also für die Synchronisation verwendet werden. Die Zuordnung zwischen den Parametern und den Komponenten erfolgt über die Namen. Die Komponenten sind wie die Parameter benannt, deren Werte sie enthalten. Z.B. gilt für folgende 'SynPCB'-Deklaration

```
SynPCB = RECORD operation: (append, remove, delete);  
          elem: elemtyp;  
          prio: priotyp;
```

END;

- Der zugeordnete abstrakte Datentyp stellt drei Operationen ('append', 'remove', 'delete') zur Verfügung.
- 'elem' ist ein Parameter vom Typ 'elemtyp' der Operationen, dessen Wert die Synchronisation beeinflusst.
- Analog 'prio'.

In IMAGAD wird das Konzept des Synchronisationskontrollblocks erweitert, da auch Informationsflußeinschränkungen von Parameterwerten abhängen. Dazu wird der Typ 'ACTIVITY' eingeführt. Der Typ ist abhängig vom umgebenden abstrakten Datentyp, kann aber aus den 'INF'-, 'COMP'- und 'PRIO'-Bedingungen automatisch erzeugt werden. Auch die Operationsnamen sind bestimmbar. Abb. 52 zeigt einen Synchronisations- und einen Schutzteil eines abstrakten Typs und den zugeordneten Aktivitätstyp. Ist eine Schutzbedingung (Informationsflußbedingung, Zugriffseinschränkung) abhängig vom Objektzustand oder von Parameterwerten, müssen diese mit 'PROT' attribuiert werden. Dies ist nur mit Parametern sichtbarer Operationen oder Komponenten der Repräsentation möglich. Objekte vom Typ 'ACTIVITY' heißen auch Aktivitätskontrollblock.

Übergabe von Schutzparametern

Die Korrektheit eines implementierten Schutzproblems hängt stark von der Versorgung der Schutzparameter ab. Sind diese für den Aufrufer einer Operation frei zugänglich, so können durch geeignete Vorbesetzung der Schutzparameter alle von den Parametern abhängigen Einschränkungen eines Datentyps umgangen werden. Der Aufrufmechanismus muß deshalb Schutzparameter bei der Parameterübergabe besonders berücksichtigen.

Eine Möglichkeit zur speziellen Behandlung von Schutzparametern bietet die Verwendung von Hardware-Registern. Häufig wird der Name des laufenden Prozesses oder eine zugeordnete Benutzerklasse (MULTICS, ADEPT-50) in Registern hinterlegt. Zur Nut-

SYN

```
COMP(a1,a2: ACTIVITY) <=> (a1.operation=insert) AND
                             (a2.operation=remove);
```

PROTECTION

```
PROT class;
INF(a1,a2: ACTIVITY) <=> (a1.operation=insert) AND
                           (a2.operation=remove) AND (a1.class>=a2.class);
```

Sich ergebende Aktivitätsstruktur:

```
ACTIVITY = RECORD operation: (insert,remove);
               class: (unbeschaenkt,vertraulich,geheim,streng_geheim)
            END;
```

Abb. 52: Aktivitätsstruktur bei vorgegebener 'INF'- und 'COMP'-Bedingung (Der abstrakte Typ stellt die Operationen 'remove' und 'insert' zur Verfügung.)

zung solcher Register in IMAGAD müssen diese dem Benutzer in einer sprachlichen Repräsentation zur Verfügung stehen. Um die Schutzkonstrukte nicht von einem konkreten Mechanismus abhängig zu machen, bietet es sich an, die vorhandenen Mechanismen einer konkreten Maschine in Form abstrakter Datentypen zur Implementation von Schutzmaßnahmen den anderen Datentypen zur Verfügung zu stellen. Der Schutz der Parameter wäre dann durch die Einschränkungen des Parametertyps gewährleistet. Da die abstrakten Schutzbedingungen nicht automatisch in konkrete Mechanismen übersetzt werden können, wäre es Aufgabe des Programmierers, die importierten Schutztypen in den Bedingungen passend zu verwenden. Stehen z.B. auf einer Maschine Capabilities zur Verfügung (System/38), so müßte ein abstrakter Datentyp 'CAPABILITIES' implementiert werden, der den Capability-Mechanismus verwendet. Im Betriebssystem PSOS (/NEUM77/) wird eine Hierarchie abstrakter Datentypen auf dem Capability-Typ aufgebaut. Der Abstraktionsmechanismus wird nicht auf sprachlicher Ebene (IMAGAD), sondern auf Betriebssystemebene, wiederum in Form eines abstrakten Typs, für Benutzer zugänglich gemacht. Die sprachliche Repräsentation in IMAGAD ist dieser Vorgehensweise überlegen, da viele notwendige Prüfroutinen im Übersetzer enthalten sind und der Programmierer weder auf Implementationsdetails abstrakter Typen Rücksicht nehmen muß, noch deren Kenntnis ausnutzen kann.

4.3.2 Zugriffseinschränkungen

Das 'PRE'-Konstrukt definiert Ausführungsbedingungen der jeweiligen Operation. Jeder Bedingung wird eine Fehlermeldung zugeordnet, die bei einem Operationsabbruch aufgrund der nichterfüllten Bedingungen als impliziter Ausgabeparameter vorliegt. Der Übergabemodus hängt vom verwendeten Aufrufmechanismus ab und wird nicht weiter erklärt.

Da eine 'ACC'-Bedingung, je nach Zuordnung, als 'NBL'- oder als 'PRE'-Bedingung interpretiert wird, sind keine speziellen Maßnahmen zu ihrer Übersetzung notwendig. Jede Operation erhält einen Vorspann zur Überprüfung der angegebenen Bedingungen. 'PRE'-Bedingungen und 'NBL'-Bedingungen jeder blockierten Operation müssen bei Änderungen des Objektzustands überprüft werden, um die Aktivität eventuell zu deblockieren. Um die Überprüfungen auf ein notwendiges Minimum zu reduzieren, sollten nur Bedingungen solcher Aktivitäten untersucht werden, deren Werte von laufenden Aktivitäten geändert worden sind. Dazu sind die in den Bedingungen abgefragten Objektkomponenten für jede Operation zu analysieren, die im weiteren in der Menge 'DEPENDSET' (Abhängigkeitsmenge) zusammengefaßt werden. 'DEPENDSET' ist den 'NBL'- und 'PRE'-Bedingungen der Operationen zugeordnet und kann wie in Bild 53 deklariert werden. Der nächste Abschnitt zeigt, daß 'DEPENDSET' auch für 'EFFECTS'-Bedingungen definiert werden kann. Ebenso sind für jede Operation die darin geänderten Komponenten des abstrakten Objekts zu analysieren ('CHANGESET', Änderungsmenge, Abb.53). Bei Beendigung jeder laufenden Aktivität wird die Schnittmenge aus

$$\text{DEPENDSET}[\text{PRE}] \cup \text{DEPENDSET}[\text{NBL}]$$

der Aktivität mit den Mengen 'CHANGESET' aller blockierten Aktivitäten gebildet. Dann brauchen nur die 'NBL'-Bedingungen der Aktivitäten untersucht zu werden, für die gilt (siehe Algorithmus, Abb. 54):

$$\text{DEPENDSET} \cap \text{CHANGESET} \neq \{\}$$

```
Komponenten = (kompname1,...,KompnameN);
              (* Komponenten der Repraesentation des betrachteten Typs *)

DEPENDSET: ARRAY[1..CARD(Condset),0..CARD(Operationen)-1]
           OF SET OF Komponenten;

CHANGESET: ARRAY[0..CARD(Operationen)-1] OF SET OF Komponenten;

Operationen: SET OF (opname1,...opnameN);
              (* gegeben durch Datentyp *)

Condset: SET OF (PRE,NBL,EFFECTS);
```

Abb. 53: Deklaration von 'DEPENDSET' und 'CHANGESET'

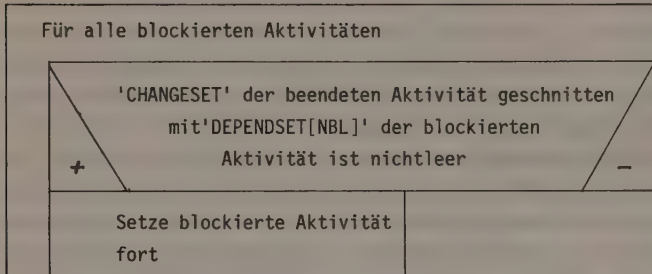


Abb. 54: Optimierung der Deblockierung bei Aktivitätsende

Berechnung von 'CHANGESET' und 'DEPENDSET'

In einer ersten Näherung sollen die Mengen 'CHANGESET' und 'DEPENDSET' bei der Übersetzung berechnet werden. Damit werden mehr Deblockierungen erzeugt als dynamisch notwendig sind, wie aus dem Beispiel der Abb.55 ersichtlich ist. In Abhängigkeit vom Wert der Variablen 'IN' wird die Objektkomponente 'K1' bzw. 'K2' geändert. Bei dynamischer Berechnung von 'CHANGESET' gilt:

CHANGESET = {K1} EXOR CHANGESET = {K2}

Statisch gilt immer:

CHANGESET = {K1,K2}

Die statische Berechnung der Mengen reduziert aber den dynamisch notwendigen Aufwand bei der Berechnung. Außerdem können bei Verwendung bekannter Techniken zur Datenflußanalyse (siehe

z.B. /MUCH81/) die statisch berechneten Mengen relativ gut an die dynamisch berechneten angenähert werden.

```
PROCEDURE p(in: BOOLEAN; par: INTEGER; VAR k1,k2: INTEGER);
```

```
  BEGIN
```

```
    IF in THEN k1:= par ELSE k2:= par END
```

```
  END p;
```

Abb. 55: Beispiel zur Berechnung von 'CHANGESET'

4.3.3 Dynamische Informationsflußanalyse

Zur Implementation von 'INF'-Einschränkungen werden sowohl Eigenschaften des inf-Prädikats als auch Eigenschaften abstrakter Datentypen ausgenutzt. Wenn die Implementation keine versteckten Kanäle enthält, die über 'NBL_PAR', 'PRE_PAR' und 'INF_PAR' (s.u.) hinausgehen, so läßt sich der maximal mögliche Informationsfluß auf den versteckten Kanal 'INF_PAR' beschränken. Diese Beschränkung ist möglich, weil für die Spezifikation eines abstrakten Datentyps die Nebenwirkungsfreiheit gefordert wird. Das inf-Prädikat wird ausgenutzt, da Informationsfluß nur zwischen aufeinanderfolgenden Änderungs- und Abfrageoperationen untersucht werden muß. Die Transitivität von allgemeinen Informationsflüssen ist dazu notwendige Voraussetzung. Die Gültigkeit der Transitivität muß außerhalb eines abstrakten Typs überwacht werden, da innerhalb nur die Beziehungen zwischen der aktuellen Aktivitätsmenge überwacht werden können. Die Transitivität wäre verletzt, wenn die Änderung des abstrakten Objekts aol und die Abfrage des abstrakten Objekts ao2 ($aol \neq ao2$) zu Informationsfluß zwischen ändernden und abfragenden Operationen führte.

Ähnlich wie bei der Überprüfung von 'PRE'- und 'NBL'-Bedingungen erfordert auch die Informationsflußkontrolle statische Maßnahmen (zur Übersetzungszeit) und dynamische Maßnahmen (zur Laufzeit). Zunächst sollen die Laufzeitmaßnahmen beschrieben werden.

Zu Beginn jeder Operation wird 'CHANGESSET' untersucht. 'DEPENDSET[EFFECTS]' enthält alle im 'EFFECTS'-Teil einer Operation abgefragten abstrakten Komponenten.

$DEPENDSET[EFFECTS] \cup DEPENDSET[NBL] \cup DEPENDSET[PRE]$ gibt alle in einer Operation abgefragten Komponenten an (Deklaration siehe Abb.53). Ist 'DEPENDSET[EFFECTS]' verschieden von der leeren Menge, so werden die enthaltenen Komponenten mit dem Aktivitätskontrollblock der laufenden Aktivität entsprechend den angegebenen 'INF'-Bedingungen verglichen. Ist die Bedingung erfüllt, so liegt kein verbotener Informationsfluß vor und die Operation kann (abhängig von 'PRE'- und 'NBL'-Bedingung) ausgeführt werden. Andernfalls wird die Operation mit einem Eintrag ins Logbuch abgebrochen. Ist die Operation ausführbar ('PRE'- und 'NBL'-Bedingung erfüllt), so werden die in 'CHANGESSET' enthaltenen Komponenten mit der Information aus dem Aktivitätskontrollblock der Operation versehen, um nachfolgende, abfragende Aktivitäten auf korrekten Informationsfluß überprüfen zu können. Ebenso wie der Wert der 'PRE'-Bedingung sich nach jeder Blockierung geändert haben kann, kann sich auch der Wert der 'INF'-Bedingung einer Aktivität während der Blockierung ändern. Deshalb muß auch die 'INF'-Bedingung nach jeder Deblockierung neu ausgewertet werden. Durch die Verwendung von Abhängigkeitsmengen ('DEPENDSET') lassen sich die notwendigen Auswertungen reduzieren, wenn für die 'PRE'- und 'NBL'-Bedingung jeder Operation jeweils eine Abhängigkeitsmenge berechnet wird. Die Differenzierung von 'DEPENDSET' anhand der Bedingungen im 'PRE'-, 'NBL'- und 'EFFECTS'-Teil erlaubt auch die Berücksichtigung der versteckten Kanäle 'PRE_PAR' und 'NBL_PAR', da für eine Abfrage von 'NBL_PAR' alle Komponenten verwendet werden müssen, von denen die 'NBL'-Bedingung abhängt (analog 'PRE_PAR').

Abb. 56 zeigt zunächst die Informationsflußprüfung bei Beginn einer Operation. Alle abgefragten Komponenten ('DEPENDSET') der

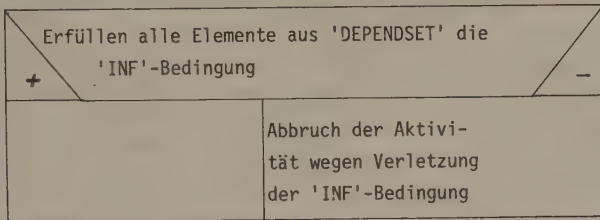


Abb. 56: Informationsflußprüfung bei Operationsbeginn

aufgerufenen Operation werden mit der 'INF'-Bedingung untersucht. Die Operation kann ausgeführt werden, wenn die 'INF'-Bedingung erfüllt ist. Abb.57 zeigt, wie die geänderten Komponenten vor Beginn einer Operation mit den Informationsflußinformationen versehen werden. Diese Informationen werden dann zur

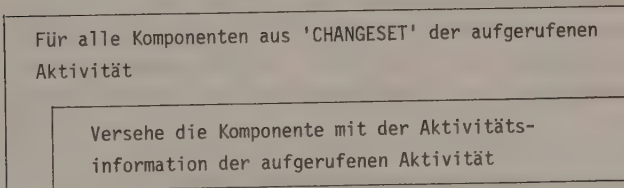


Abb. 57: Setzen der Informationsflußinformationen

Überprüfung der 'INF'-Bedingung verwendet. Schließlich enthält Abb.58 die bei Beginn einer Operation oder bei ihrer Deblockierung auszuführenden Überprüfungen bei vorhandenen 'INF'-, 'SYN'-, 'PRE'- und 'NBL'-Bedingungen.

Die dynamische Bestimmung der Mengen 'CHANGESET' und 'DEPENDSET' für jede Aktivität könnte den Feinheitgrad der untersuchten Informationsflüsse erhöhen. Bei der Definition der Synchronisationseinschränkungen müßte aber auf die dynamischen Änderungen der Informationsflußinformation Rücksicht genommen werden.

-		Informationsflußbedingung verletzt			+
-		'PRE'-Bedingung verletzt			+
-		'NBL'-Bedingung verletzt		+	Abbruch der Aktivität, eventuell Eintrag in das Logbuch
-		'COMP'-Bedingung nicht erfüllt		+	
-		Aktivität hat nicht höchste Priorität		+	
-		Aktivität wird "wartend"		+	
Setzen der Informationsflußinformationen, Aktivität wird "laufend"		Aktivität wird "laufbereit"		Blockierung der Aktivität	Abbruch der Aktivität, Ausgabe der Nummer der verletzten 'PRE'-Bedingung als Fehlermeldung

Abb. 58: Überprüfungen bei Operationsbeginn

Dies widerspricht dem Charakter der Schutzkonstrukte, deren Implementation keinen Einfluß auf die Implementation des abstrakten Datentyps haben soll.

Zur statischen Berechnung der Mengen 'CHANGESET' und 'DEPENDSET' müssen die geänderten Komponenten eines abstrakten Objekts und die abgefragten Komponenten, die sich in Ausgabeparametern zeigen, berechnet werden. Der 'INF_PAR'-Parameter braucht nicht berücksichtigt zu werden, da bei einem verbotenen Informationsfluß dieser Kanal (dynamisch) nicht geschlossen werden kann. Seine Berücksichtigung kann aber zu Eintragungen in ein Logbuch für verbotene Informationsflüsse verwendet werden.

Ein Nachteil der beschriebenen Vorgehensweise ist, daß nie

die Änderung einer Variablen zu einem Abbruch führt, sondern nur deren Abfrage in einem 'PRE'-, 'NBL'- oder 'EFFECTS'-Teil. Dies liegt daran, daß Informationsfluß als Folge von Änderungen und Abfragen definiert ist. Bei Vorliegen einer Änderung ist noch nicht die semantische Abfrage entscheidbar, da eine entsprechende Leseoperation nicht unbedingt folgen muß. Dadurch ist es möglich, daß ein böswilliger Schreiber "nicht erlaubte" Änderungen ausführt, die aber erst bei einem entsprechenden Lesevorgang bemerkt werden. Solche Effekte müssen durch flankierende Zugriffsschutzmaßnahmen unterbunden werden.

4.4 Beispiele

Im folgenden soll an den beiden abstrakten Datentypen 'pnezev' und 'ppptisch', die in Kapitel 3 spezifiziert wurden, die Funktionsweise der 'INF'- und 'ACC'-Überprüfungen deutlich gemacht werden (und die Verwendung der angegebenen Laufzeitroutinen).

Der abstrakte Typ 'pnezev'

Die Implementation des Typs 'pnezev' zeigt Abb.59. An dieser Stelle wird nicht näher auf Konstrukte der Implementationsprache, wie z.B. das 'WHERE'-Konstrukt, eingegangen, sondern die dynamische Abarbeitung von aufgerufenen Operationen des Datentyps steht im Vordergrund. Dazu wird eine Folge von Operationsaufrufen am Objekt 'P' vom Typ 'pnezev' betrachtet, das wie folgt deklariert ist:

```
P: pnezev(INTEGER,2);
```

Abb. 60 zeigt die Mengen 'CHANGESET' und 'DEPENDSET' des Datentyps.

Der Aufruf 'P.remove(C1) -> E1' führt zu keinen Objektveränderungen und wird zunächst blockiert, da 'P' noch kein zu ent-

```

ABSTRACT TYPE IMPLEMENTATION pnezev
    (t: TYPE; max_elements: CARDINAL WHERE max_elements>0);

CONCRETE REPRESENTATION
    elements: ARRAY[0..max_elements-1] OF t;
    in,out: CARDINAL INIT in:= 0; out:= 0 END;

TYPE
    classification =(unbeschraenkt,vertraulich,geheim,streng_geheim);

SYN
    COMP(a1,a2: ACTIVITY) <=> (a1.operation=insert) AND
                                (a2.operation=remove) AND (in-out>1);

PROTECTION
    PROT prot_class;
    NINF(a1,a2: ACTIVITY) <=> (a1.operation=insert) AND
                                (a2.operation=remove) AND
                                (a1.prot_class > a2.prot_class);

OPERATIONS

PROCEDURE insert(prot_class: classification; elem:t);
    NBL in-out<max_elements;
    BEGIN
        elements[in MOD max_elements]:= elem;
        INC(in);
    END insert;

PROCEDURE remove(prot_class: classification; VAR elem:t);
    NBL in>out;
    BEGIN
        elem:= elements[out MOD max_elements];
        INC(out);
    END remove;

END pnezev.

```

Abb. 59: Implementation des Typs 'pnezev'

fernendes Element enthält. Deshalb soll auch noch keine Informationsflußverletzung vorliegen. Prinzipiell kann aus der Blockierung der Operation Informationsfluß entstehen ('NBL_PAR'), der in 'DEPENDSET' berücksichtigt ist, da aus der Blockierung Aussagen über die Länge des Objekts 'P' abgeleitet werden können. Ein zweiter Aufruf 'P.remove(C2) -> E2' wird analog zum ersten Aufruf behandelt.

Die erste Einfügung 'P.insert(C3,E3)' wird nicht blockiert und kann 'E3' in das abstrakte Objekt eintragen. Eine zweite Einfüge-Operation 'P.insert(C4,E4)' wird wegen der Nichtver-

'DEPENDSET'	insert	remove
[PRE]	%	%
[NBL]	in,out	in,out
[EFFECTS]	%	pnezev[out]
'CHANGESET'	in pnezev[in]	out

Abb. 60: 'CHANGESET' und 'DEPENDSET' des Typs 'nezev'

träglichkeit "wartend", aber nicht "blockiert", da die maximale Länge des abstrakten Objekts noch nicht erreicht ist. Da zwischen zwei Einfügungen Informationen fließen dürfen, ist die Operation nicht abubrechen.

Nach Beendigung der ersten Einfügung kann die zweite Einfügung "laufend" werden. Die Nichtblockierungsbedingung ist noch nicht erfüllt. Außerdem wird die erste 'remove'-Operation "laufend", wobei die Informationsflußbedingung erneut zu untersuchen ist, da das abstrakte Objekt ein Element enthält. Die Operation ist nur ausführbar, wenn gilt: $C1 \geq C3$. Unter der Annahme, daß die Bedingung erfüllt ist, muß zusätzlich der Informationsfluß über den Parameter 'NBL_PAR' kontrolliert werden. Deshalb muß zusätzlich gelten: $C1 \geq C4$. Gilt auch diese Einschränkung, so kann die 'remove'-Operation ausgeführt werden. Dabei wird die Klasse des abstrakten Objekts zu $C1$. Schließlich kann auch die zweite 'remove'-Operation deblockiert werden. Dabei ist zu überprüfen: $C2 \geq C4$. Da die vorhergehende 'remove'-Operation die Anzahl der Einträge des abstrakten Objekts verändert hat, muß zusätzlich gelten: $C2 \geq C1$. Sollen die versteckten Kanäle nicht berücksichtigt werden, so genügt die Implementation des abstrakten Typs 'apnezev'. Dort wird der Informationsfluß zwischen zwei Klassen als 'ACC'-Bedingung formuliert.

Das abstrakte Typ 'pphtisch'

Im spezifizierten Typ 'pphtisch' wird durch einen Mechanismus zur Rechtevergabe die korrekte Belegung und Rückgabe von Gabeln in einem m-Philosophenproblem ($m \geq 2$) garantiert. Die Lösung des Problems beschreibt Kapitel 3. Abb.61 enthält die Implementation. Hier soll durch Betrachtung einer Aufruffolge an einem Objekt des Typs beispielhaft die unterschiedliche Wirkungsweise der 'PRE'- und 'NBL'-Bedingungen erläutert werden. Außerdem werden die Rechte mit Hilfe des Typs 'Rechtevektor' implementiert. Der Parameter gibt die Anzahl der Rechte eines konkreten Rechtevektors an. 'Rechtevektor' sei ein eigener abstrakter Typ, der eine besonders effiziente Implementation für Rechtevergleiche enthält, mit den Operationen 'Rechte_setzen' (belegt die in einem Rechtevektor enthaltenen Rechte) und 'Rechtevergleich' (überprüft, ob ein Rechtevektor bestimmte Rechte enthält).

Zunächst wird ein Objekt vom Typ 'pphtisch' deklariert:

```
philosophen: pphtisch(3);
```

Mit der ersten 'belegen'-Operation

```
philosophen.belegen(2);
```

werden die Gabeln der Philosophen 1 und 2 belegt. Philosoph 2 bekommt das Recht 'schlaf'. Dafür darf er keine Gabeln mehr belegen, da die Rechte 'linke_gabel' und 'rechte_gabel' entfernt werden. Mit der zweiten 'belegen'-Operation

```
philosophen.belegen(1);
```

wird der Philosoph 1 blockiert, da sein Nachbar ist.

Eine Operation 'philosophen.freigeben(1);' wäre abgebrochen worden, da Philosoph 1 das Recht 'schlaf' nicht hat. Er kann nur mit diesem Recht Gabeln freigeben.

Der dritte Operationsaufruf 'philosophen.freigeben(2);' bewirkt, daß Philosoph 0 und 1 ein Gabelrecht erhalten (es kann bereits vorhanden sein). Die Gabeln werden freigegeben, so daß der blockierte Philosoph 1 deblockiert werden kann. Da auch die beiden Rechte 'linke_gabel' und 'rechte_gabel' in seiner Rechtemenge enthalten sind, ist die 'belegen'-Operation ausführbar,

```

ABSTRACT TYPE IMPLEMENTATION pphtisch(m: CARDINAL WHERE m>=2);

CONCRETE REPRESENTATION
  gabeln: SET OF [0..m-1] INIT gabeln:= {0..m-1};
  ph_rechte: ARRAY[0..m-1] OF Rechtevektor(3)
    INIT FOR i:=0 TO m-1 DO
      ph_rechte[i].Rechte_setzen({ORD(rechte_gabel),
                                ORD(link_gabel)});

VAR i: INTEGER;

IMPORT
  FROM Rechtevektor:
    PROCEDURE Rechte_setzen(i: SET OF CARDINAL);
    PROCEDURE Rechtevergleich(i: SET OF CARDINAL): BOOLEAN;

TYPE
  Rechte = (linke_gabel, rechte_gabel, schlaf);

PROTECTION
  PROT ph_rechte, pnr;

OPERATIONS

PROCEDURE belegen(pnr: CARDINAL);
  NBL ((pnr+1) MOD m IN gabeln) AND (pnr IN gabeln);
  BEGIN
    gabeln:= gabeln-((pnr+1) MOD m, pnr);
    ph_rechte[pnr].Rechte_setzen({ORD(schlaf)});
  END belegen;

PROCEDURE freigeben(pnr: CARDINAL);
  PRE ACC <=> NOT ph_rechte.Rechtevergleich({ORD(schlaf)});
  BEGIN
    gabeln:= gabeln+((pnr+1) MOD m, pnr);
    ph_rechte[(pnr+1) MOD m].Rechte_setzen({ORD(link_gabel)});
    ph_rechte[(pnr-1) MOD m].Rechte_setzen({ORD(rechte_gabel)});
    ph_rechte[pnr].Rechte_setzen({ORD(schlaf)});
  END freigeben;

END pphtisch.

```

Abb. 61: Implementation des Typs 'pphtisch'

die Gabeln werden belegt und Philosoph 1 bekommt das Recht 'schlaf'. Der vierte Aufruf

```
philosophen.belegen(2);
```

muß zunächst blockiert werden, da Philosoph 1 noch ißt. Auch nach dem Operationsaufruf

```
philosophen.freigeben(1);
```

kann Philosoph 2 nicht deblockiert werden, da er zwar das Recht 'linke_gabel', aber nicht das Recht 'rechte_gabel' hat. Er muß

solange warten (ebenso Philosoph 1), bis Philosoph 0 einmal gegessen hat:

```
philosophen.belegen(0);  
philosophen.freigeben(0);
```

Dann kann zunächst Philosoph 2 essen. Nachdem er fertig ist, kann auch Philosoph 1 essen, da dieser dann das Recht 'rechte_gabel' wieder hat.

4.5 VAX-Realisierung

Die vorgeschlagenen Sprachkonstrukte zur Realisierung abstrakter Datentypen müßten mit Hilfe eines eigenen Compilers übersetzt werden, um die Möglichkeiten der Typerweiterung ausnutzen zu können. Die gemeinsame Bearbeitung von Objekten beliebigen Typs wird kaum von Betriebssystemen oder Compilern unterstützt. Um mit möglichst wenig Aufwand zu einer Testversion des Systems zu gelangen, wird folgende Vorgehensweise vorgeschlagen (In /RUPP83/ ist das Konzept bereits für Synchronisationskonstrukte verwirklicht.):

- Entwicklung eines Präprozessors zur Abbildung der vorgeschlagenen Sprachkonstrukte auf Modula-2
- Verwendung von Modula-2 als Wirtssprache wegen des darin enthaltenen Modularisierungskonzepts
- Realisierung gemeinsamer Objekte mit Hilfe gemeinsamer Speicherbereiche des Systems VMS ("shareable images")

Mit dieser Vorgehensweise sind verschiedene Nachteile verbunden. Der zusätzliche Lauf des Präprozessors verzögert die Übersetzungszeit, insbesondere da das Quellprogramm zweimal durchlaufen werden muß. Dieser Nachteil kann in einer Testversion vernachlässigt werden. Wie aus der Beschreibung der abstrakten Datentypen ersichtlich ist, ist das vorgeschlagene Abstraktionskonzept nicht mit dem Modul-Konzept vereinbar. Abstrakte Datentypen sind nicht parametrisierbar und sie lassen sich nicht verbergen, so daß Zugriffe auf die interne Struktur eines

Moduls nicht ausgeschlossen werden können. Die in Modula-2 enthaltene Typprüfung beschränkt sich auf Typkonsistenz in Wertzuweisungen und Ausdrücken. Dieser Nachteil wird durch die Annahme gerechtfertigt, daß in der Testversion zunächst nur kooperative Benutzer arbeiten, die die nur implizit vorhandenen Typrestriktionen beachten. Der Zugriff auf gemeinsame Objekte kann nicht in der vorgeschlagenen allgemeinen Form erfolgen. Die Einrichtung gemeinsamer Speicherbereiche erfordert spezielle Betriebssystemaufrufe, die nicht allgemein für Benutzerprogramme zugänglich gemacht werden dürfen. Der Aufbau gemeinsamer Speicherbereiche und ihre Größe sind nicht dynamisch änderbar. Auch die Sprache Modula-2 bietet keine Hilfsmittel, zwischen gemeinsam und einzeln benutzten Objekten zu unterscheiden. Dazu müssen mit dem Präprozessor übersetzte Datentypen speziell gebunden werden.

Der Präprozessor

In einem Vorlauf werden die speziellen Sprachkonstrukte zur Beschreibung geschützter, asynchroner abstrakter Datentypen in ein Modula-2-Modul übersetzt. Dazu ist einerseits der Typname in den 'TYPE'-Teil zu übernehmen. Da Modulnamen mit Typnamen identisch sein können, kann das Modul mit dem Namen der konkreten Repräsentation versehen werden. Besteht die konkrete Repräsentation aus mehreren Komponenten, so müssen diese zu einem 'RECORD'-Typ zusammengestellt werden, da nur so Objekte des Typs in Modula-2 vereinbart werden können. Der Synchronisationsteil enthält logische Ausdrücke, die in logische Funktionsprozeduren umgewandelt werden. Als Parameter werden jeweils zwei Aktivitätskontrollblöcke (AKB) und eventuell notwendige Objektzustände übergeben. Hängt die Synchronisation zusätzlich von Operationsparametern ab, sind diese bereits im AKB enthalten.

Bei Aufruf der Operationen muß der Synchronisationsmechanismus angestoßen werden, um die formulierten Bedingungen entsprechend der vorgegebenen Semantik in Operationseinschränkungen umzusetzen. Ebenso muß am Ende einer Operation zum Synchronisa-

tionsmechanismus verzweigt werden, um den eventuell veränderten Zustand bei der Synchronisation zu berücksichtigen. Die Synchronisationseinschränkungen werden auf die von VMS bereitgestellten Semaphore zurückgeführt. Da für die Prozessorzuteilung unter VMS kein Unterschied zwischen "wartend" und "blockiert" gemacht wird, werden alle nichtbereiten und nichtlaufenden Aktivitäten in einer Warteschlange angeordnet. Mögliche Optimierungen sind darin nicht berücksichtigt. Die Unterscheidung zwischen wartenden und blockierten Prozessen entfällt, da beim Beenden einer Aktivität alle nichtbereiten Prozesse untersucht werden müssen, ob sich der Wert der Blockadebedingung geändert hat. Die Unterscheidung zwischen "bereit" und "wartend" braucht nicht explizit gemacht zu werden, da die Änderungen der laufenden Prozesse automatisch die Aufteilung zwischen bereiten und wartenden Prozessen ändert. Im Extremfall, bei leerer Menge laufender Prozesse, ist auch die Menge der wartenden leer (aufgrund der Synchronisationseigenschaften).

Der Teil 'PROTECTION' kann die Kennzeichnung von Schutzobjekten und 'INF'-Einschränkungen enthalten. Die Schutzobjekte werden zur Überprüfung der formulierten Schutzbedingungen verwendet. Das 'INF'-Konstrukt erfordert einerseits die Berechnung der Mengen 'CHANGESET' und 'DEPENDSET'. Andererseits wird der Vorspann jeder Operation um die Überprüfung der 'INF'-Bedingungen erweitert.

In den Operationen muß der 'PRE'-Teil berücksichtigt werden. Dazu wird die Parameterliste jeder Operation um einen Ausgabe-parameter

STATUS: CARDINAL;

erweitert, der angibt, welche der 'PRE'-Bedingungen zum Operationsabbruch führt. Zur Optimierung der zu untersuchenden Bedingungen können die Mengen 'DEPENDSET' und 'CHANGESET' mit Bitvektoren implementiert werden.

Der Postprozessor

Durch die Abgeschlossenheit des Modula-2-Systems, das Übersetzer, Binder und Lader enthält, ist es nicht möglich, gemeinsame Speicherbereiche aus Modula-2-Objektcode zu erzeugen. Deshalb wird der Bindecode in MACRO-Assembler-Quellsprache umgewandelt, die dann assembliert und gebunden wird. Der so erzeugte Ladecode kann zum Aufbau gemeinsamer Speicherbereiche verwendet werden.

Das Laufzeitsystem

In das Laufzeitsystem sind Programme aufzunehmen, die den Synchronisations- und Schutzmechanismus realisieren (siehe Algorithmen, Abschnitt 3.3). Der Präprozessor erzeugt lediglich Aufrufe an das Laufzeitsystem. Daraus ergeben sich zwei Vorteile. Einmal bleiben die erzeugten Programme unabhängig von Änderungen der Mechanismen, sie müssen nur neu gebunden werden. Andererseits können die Mechanismen zentral an Änderungen des Betriebssystems angepaßt werden.

Abstrakte Objekte

Die vorgestellte Implementationsweise abstrakter Datentypen erfordert die explizite Erzeugung und Zerstörung abstrakter Objekte. Der notwendige Speicherplatz muß vom zugehörigen Datentyp verwaltet werden. Für den Zugriff mehrerer VMS-Prozesse auf ein gemeinsames abstraktes Objekt werden die Objekte analog zur Dateiverwaltung benannt, sind also über ASCII-Zeichenketten zugänglich.

Da der Aufwand zum Zugriff auf abstrakte Objekte von der Zahl der zugreifenden Prozesse abhängt, ist es sinnvoll, bei der Erzeugung der Objekte die maximale Anzahl der Zugriffe vorzugeben. Ist nur ein Prozeß zugriffsberechtigt, können 'INF', 'COMP'- und 'PRIO'-Bedingungen entfallen. Die 'NBL'-Bedingungen müssen wie 'PRE'-Bedingungen behandelt werden, so daß der Prozeß daran nicht unendlich lange blockiert wird.

5 Verifikation spezifizierter und implementierter Schutzprobleme

Die Spezifikationssprache SPAGAT unterstützt die problemnahe Beschreibung abstrakter Datentypen. Wesentlich ist, daß durch Einführung von allgemeinen Synchronisations- und Schutzkonstrukten auch Synchronisations- und Schutzprobleme der Datentypen spezifizierbar sind. Die Verwendung prädikatenlogischer Ausdrücke erlaubt eine formale und exakte Problembeschreibung, verbietet aber nicht die Spezifizierung unentscheidbarer Probleme. Die Sprache IMAGAD zur Implementation abstrakter Datentypen verwendet Elemente aus sequentiellen Programmiersprachen. Synchronisations- und Schutzeigenschaften werden durch spezielle Konstrukte automatisch auf einen Schutz- und Synchronisationsmechanismus zurückgeführt, der die aufrufenden Prozesse entsprechend der vorgegebenen Semantik der Konstrukte steuert. Die Sprache verbietet aber nicht die Implementierung nicht terminierender Programme oder sinnloser Schutz- und Synchronisationseigenschaften. Erst über ein Bindeglied zwischen der Spezifikation und der Implementation eines abstrakten Typs kann die Entscheidbarkeit einer Spezifikation und die Korrektheit der Typimplementation überprüft werden.

Als Bindeglied dient eine Funktion, die den Wertebereich der konkreten Repräsentation KR auf den Wertebereich der abstrakten Repräsentation AR surjektiv abbildet. Die Funktion heißt Repräsentationsfunktion REP. Sie kann partiell sein, da nicht jeder Zustand der konkreten Repräsentation ein abstraktes Pendant haben muß. Die Funktion kann zu einer totalen Funktion REPT ergänzt werden, wenn man jeden nicht in $\text{DOM}(\text{REP})$ enthaltenen Zustand auf einen speziellen Zustand $\text{Error} \in \text{RAN}(\text{REPT})$ abbildet. Auf diese Weise können unerwartete Zustände der Implementation, wie sie z.B. bei Hardware-Fehlern entstehen, berücksichtigt werden. Im weiteren soll nur die partielle Funktion REP betrachtet werden. Die Surjektivität ist notwendig, da sonst für spezifizierte Zustände der abstrakten Repräsentation keine konkreten Zustände existierten, also die Implementation unvollständig wäre. Grundlage der im weiteren zu entwickelnden Verifikationsbedingungen ist folgende Abb.62, die den Zusammenhang

zwischen konkreter und abstrakter Repräsentation aufzeigt. Der Zustand AO eines abstrakten Objekts geht durch die Operation op in den Zustand AO' über. Dann muß der konkrete Zustand KO in den konkreten Zustand KO' übergehen, wenn das zu op gehörige Implementationsprogramm Imp(op) ausgeführt wird. Innerhalb des Implementationsprogramms Imp(op) sind konkrete Zustände erreichbar, die nicht als abstrakte Zustandsübergänge in Erscheinung treten. Die im folgenden anzugebenden Verifikationsbedingungen fordern die Kommutativität des Diagramms, d.h.

$$op(Rep(KO)) = Rep(Imp(op)(KO))$$

Der folgende erste Abschnitt geht zunächst auf Konsistenzbe-

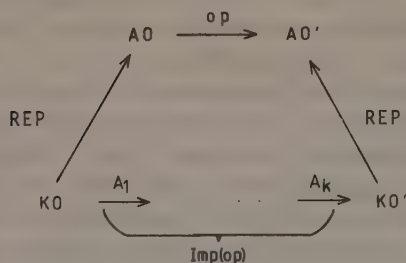


Abb. 62: Konkrete und abstrakte Zustandsübergänge

dingungen der Spezifikation, unabhängig von einer speziellen Implementation ein. Konsistenzverletzungen können bei Verwendung des 'INF'-Konstrukts auftreten, wenn versteckte Kanäle berücksichtigt werden müssen. Durch Invarianzeigenschaften, die ein spezifiziertes Problem erfüllen muß, ist es möglich, sowohl versteckte Kanäle der Spezifikation als auch andere Eigenschaften des Typs zu beschreiben und die Spezifikation bez. dieser Eigenschaften zu überprüfen.

Im zweiten Abschnitt werden mit Hilfe der REP-Funktion Korrektheitsbedingungen der Implementation abgeleitet. Diese sind für jedes Spezifikations-, Implementationspaar eines abstrakten Typs nachzuweisen. Unter der Voraussetzung, daß die Spezifikation einen abstrakten Typ korrekt beschreibt, ist dann die Korrektheit der Implementation formal beweisbar. Dazu müssen die im zweiten Abschnitt aus der Repräsentation abgeleiteten Vor-

und Nachbedingungen (Bedingungen der implementierten Operationen) von den Implementationsprogrammen erfüllt werden. Dabei kann die bekannte Hoaresche Verifikationsmethode angewandt werden. Die in /KERA82/ angegebenen Bedingungen (siehe Zusammenstellung in Anhang II) sind hinsichtlich ihrer Anwendbarkeit auf die Schutzkonstrukte zu erweitern. Der dritte Abschnitt geht auf Auswirkungen der Schutzkonstrukte auf die Verifikation der einzelnen Implementationsprogramme ein. Die Beispielverifikation des Typs 'pnezev' schließt das Kapitel ab.

5.1 Konsistenzbedingungen der Spezifikation

Die SPAGAT-Spezifikation eines abstrakten Datentyps erlaubt zunächst keine formalen Aussagen über die Korrektheit der Spezifikation. Es wird angenommen, daß die Spezifikation die erste formale Definition eines abstrakten Typs ist. Auf vorausgehende Methoden und Techniken zur Entwicklung einer Spezifikation wird an dieser Stelle nicht eingegangen (siehe auch /WEDE81/). Trotzdem gibt es Hilfsmittel, die die Überprüfung einer Typspezifikation gestatten. Einmal kann mit der Methode des "rapid prototyping" aus einer formalen Spezifikation ein Programm erzeugt werden, das den abstrakten Datentyp bez. der spezifizierten Eigenschaften simuliert. Anhand des erzeugten Prototyps ist empirisch zu untersuchen, inwieweit die Spezifikation der Vorstellung von dem abstrakten Typ entspricht. Zum anderen beschreibt Keramidis in /KERA82/, wie eine Typspezifikation bez. formalisierter invarianter Typeigenschaften formal untersucht werden kann. Darauf wird im folgenden genauer eingegangen.

Zur Vereinfachung der Bedingungen werden einige Abkürzungen für die verwendeten Objekte eingeführt. KO bezeichne ein konkretes Objekt der Implementation, AO sei ein abstraktes Objekt der Spezifikation. Beide sind über die Repräsentationsfunktion REP verknüpft:

$$KO \in \text{DOM}(\text{REP}), AO \in \text{RAN}(\text{REP}), AO = \text{REP}(KO).$$

Mit ANBL, APRE, ACOMP, APRIO, AACC, AEFFECTS werden die Bedingungen der zugehörigen Spezifikationskonstrukte ('NBL', 'PRE', usw.) bezeichnet; analog bezeichnen KNBL, KPRE usw. die Bedingungen der entsprechenden Implementationskonstrukte. Die Menge OPERATIONS enthält die Operationen eines abstrakten Typs. Ein Index op steht für die Bedingungen an der jeweiligen Operation op des abstrakten Typs. Die Initialisierungsbedingungen der Spezifikation bzw. die Initialisierung der Implementation werden in AINIT bzw. KINIT zusammengefaßt. (Die Abkürzungen gelten auch für die Zusammenstellung der Verifikationsbedingungen in Anhang II.)

Für einen abstrakten Datentyp lassen sich Typinvarianten (ähnlich zu Schleifeninvarianten in Programmen) angeben, die bestimmte, invariante Eigenschaften des Typs beschreiben. Die Invarianten sind beim Übergang von der informellen zur formalen Spezifikation interessant, um eine Spezifikation auf vorgegebene Eigenschaften, die zunächst als Invarianten zu formulieren sind, zu überprüfen. AI bezeichne die Invarianten der Spezifikation und KI die Invarianten der Implementation. Dann müssen für eine korrekte Spezifikation folgende Bedingungen gelten:

I AINIT(AO) ---> AI(AO)

(Jedes abstrakte Objekt erfüllt nach seiner Initialisierung die invarianten Typeigenschaften.)

II $(\forall op \in OPERATIONS) (AI(AO) \wedge APRE_{op}(AO) \wedge ANBL_{op}(AO) \wedge AEFFECTS_{op}(AO, AO') \text{ ---> } AI(AO'))$

(Ist eine Operation ausführbar und gelten die Invarianzeigenschaften, so gelten die Invarianzeigenschaften auch nach Ausführung der Operation.)

Wegen der Unteilbarkeit der abstrakten Operationen brauchen 'COMP'- und 'PRIO'-Bedingungen nicht berücksichtigt zu werden. Die Schutzzeigenschaften eines abstrakten Typs (mit Ausnahme der 'INF'-Bedingungen) sind in den Bedingungen I und II berücksichtigt, da die Schutzkomponenten in der abstrakten Repräsentation AO enthalten sind und Zugriffseinschränkungen in Form von 'NBL'- oder 'PRE'-Bedingungen in den Bedingungen I und II enthalten sind.

Spezifikation von Informationsflußeinschränkungen

Bei der Spezifikation von Informationsflußeinschränkungen können weitere Invarianzeigenschaften hinzukommen, wenn der Informationsfluß über den Parameter 'INF_PAR' eingeschränkt wird. Das bedeutet, daß die Informationsflußeinschränkungen selbst nicht zu Informationsflüssen führen dürfen. Diese Eigenschaft kann nur sichergestellt werden, wenn die 'INF'-Bedingungen nicht nur dynamisch, sondern auch statisch erfüllt sind. Wären sie nur dynamisch erfüllbar, so könnte durch Beobachtung des Zustands von 'INF_PAR' Information fließen. Ist der Informationsfluß über 'INF_PAR' nicht eingeschränkt, so genügt die dynamische Kontrolle, da bei Informationsflußverletzungen nur der Zustand von 'INF_PAR' beobachtbar ist. Diese Eigenschaft wird bei der Implementation des 'INF'-Konstrukts verwendet, da verbotene Informationsflüsse zum Operationsabbruch führen.

In die statische Überprüfung der Informationsflußeigenschaften müssen auch die versteckten Kanäle der Implementation, 'NBL_PAR' und 'PRE_PAR', einbezogen werden. Das bedeutet, daß die 'INF'-Bedingung nicht auf den 'EFFECTS'-Teil einer Operation beschränkt ist, sondern auch die im 'NBL'- und im 'PRE'-Teil auftretenden Komponenten berücksichtigen muß. Die Mengen 'DEPENDSET' und 'CHANGESET' der Implementation erleichtern die zu formulierende Verifikationsbedingung.

$$\begin{aligned} \text{III } (\forall op_1, op_2 \in \text{OPERATIONS}) (\forall y \in \text{REP}((\text{DEPENDSET}[\text{EFFECTS}, op_1] \cup \text{DEPENDSET}[\text{NBL}, op_1] \\ \cup \text{DEPENDSET}[\text{PRE}, op_1]) \cap \text{CHANGESET}[op_2])) \\ (\inf(op_2, op_1, y, B_1, B_2) < \text{---} \rightarrow \text{AINF}) \end{aligned}$$

(Zwischen je zwei Operationen op_2 und op_1 fließt höchstens dann Information, wenn die abstrakte Informationsflußeinschränkung dies erlaubt. Die Bedingungen B_1 und B_2 des inf-Prädikats ergeben sich aus der abstrakten Informationsflußbedingung AINF.)

An Bedingung III fällt die Ähnlichkeit zur Definition des 'INF'-Konstrukts in Kapitel 3 auf (Implikation statt Äquivalenz). Die Implikation ergibt sich daraus, daß die zum abstrakten Typ gehörige Betriebssystemmaschine nicht dynamisch, sondern statisch untersucht wird. Während in Kapitel 3 aus dem inf-Prädikat die Bedeutung einer formulierten 'INF'-Bedingung zu definieren war, wird hier aus der formulierten 'INF'-Bedingung (der abstrakten Invarianten) die zu beweisende Verifikationsbedingung abgeleitet.

5.2 Formale Beziehungen zwischen Spezifikation und Implementation

Zum formalen Nachweis der Korrektheit der Implementation eines abstrakten Typs bez. einer Spezifikation müssen zu verifizierende Beziehungen zwischen beiden entwickelt werden. Dazu wird die bereits in der Einleitung definierte Funktion REP verwendet. Es wird stets von der Spezifikation bzw. Implementation eines speziellen abstrakten Typs ADT gesprochen.

Schutzkomponenten

Die in der Spezifikation und Implementation enthaltenen Schutzobjekte sind Teil des konkreten und des abstrakten Zustandsraums (KO bzw. AO) und können damit in REP berücksichtigt werden. Schutzkomponenten werden nicht mit einer eigenen Repräsentationsfunktion abgebildet, um die Möglichkeit zur Implementation der abstrakten Komponenten nicht einzuschränken.

Ausführungsbedingungen

Keramidis fordert folgende Beziehung zwischen den konkreten und abstrakten Ausführungsbedingungen jeder Operation:

$$* \quad KI(KO) \wedge ANBL_{op}(AO) \longleftrightarrow KNBL_{op}(KO)$$

Damit wird ausgedrückt, daß eine konkrete Operation nur dann ausführbar ist, wenn sie auch im abstrakten Zustandsraum ausführbar ist. Die konkrete Invariante KI erleichtert den Nachweis der angegebenen Bedingung.

Prinzipiell ließe sich die Bedingung auch für geschützte Objekte übernehmen. Durch die Semantik der neu eingeführten Konstrukte 'PRE' und 'ACC' muß zwischen blockierenden ('NBL') und abbrechenden ('PRE') Bedingungen einer Operation unterschieden werden. Das 'ACC'-Konstrukt soll ebenfalls besonders hervorgehoben werden, um die Äquivalenz von konkreten und abstrakten Schutzeinschränkungen überprüfen zu können. Die folgenden Verifikationsbedingungen besagen damit nicht nur, daß die Ausführung abstrakter und konkreter Operationen bei korrespondierenden Zuständen des abstrakten und konkreten Zustandsraums möglich ist. Sie fordern darüber hinaus die Korrespondenz der einzelnen Abbruchs- und Blockierungsbedingungen, der einzelnen Abbruchsbedingungen und der einzelnen Schutzeinschränkungen.

Zunächst werden zu möglichen abstrakten Invarianten AI konkrete Invarianten KI abgeleitet, die die Beweise der einzelnen Bedingungen vereinfachen:

$$(\forall y \in \text{RAN}(\text{REP}))(y = \text{REP}(x) \longleftrightarrow (\text{AI}(y) \longleftrightarrow \text{KI}(x)))$$

Für die Initialisierung eines konkreten Objekts muß gelten:

$$\text{IV} \quad (\forall y \in \text{RAN}(\text{REP}))(y = \text{REP}(x) \longleftrightarrow (\text{KINIT}(x) \longleftrightarrow \text{AINIT}(y) \wedge \text{KI}(x)))$$

Zur Formulierung der weiteren Bedingungen werden folgende Schreibweisen vereinbart. 'PRE'-Bedingungen werden in der Reihenfolge ihres Auftretens durchnummeriert. Dies wurde bereits in Kapitel 4 beschrieben, wo für jede durch "---" getrennte Bedingung eine eigene Fehlernummer vorgesehen ist. Eine 'PRE'-Bedingung ist also zu schreiben als:

$$\text{PRE}_{op} = \langle \text{PRE}_{1,op}, \text{PRE}_{2,op}, \dots, \text{PRE}_{nop,op} \rangle$$

mit der Semantik:

$$PRE_{op} \longleftrightarrow (PRE_{1,op} \wedge PRE_{2,op} \wedge \dots \wedge PRE_{nop,op})$$

Analog gilt für die 'ACC'-Bedingungen einer Operation (sowohl aus dem 'PRE'- als auch aus dem 'NBL'-Teil):

$$ACC_{op} = \langle ACC_{1,op}, ACC_{2,op}, \dots, ACC_{mop,op} \rangle$$

$$ACC_{op} \longleftrightarrow (ACC_{1,op} \wedge ACC_{2,op} \wedge \dots \wedge ACC_{mop,op})$$

Daraus ergeben sich folgende Verifikationsbedingungen:

V ($\forall op \in OPERATIONS$)

$$\begin{aligned} & [(\forall i \in \{1, \dots, mop\}) (\forall y \in \text{RAN}(\text{REP})) (y = \text{REP}(x) \longrightarrow \\ & \quad (KI(x) \wedge ACC_{i,op}(y) \longleftrightarrow KACC_{i,op}(x))) \\ & \wedge (\forall i \in \{1, \dots, nop\}) (\forall y \in \text{RAN}(\text{REP})) (y = \text{REP}(x) \longrightarrow \\ & \quad (KI(x) \wedge APRE_{i,op}(y) \longleftrightarrow KPRE_{i,op}(x))) \\ & \wedge (\forall y \in \text{RAN}(\text{REP})) (y = \text{REP}(x) \longrightarrow (KI(x) \wedge ANBL_{op}(y) \longleftrightarrow KNBL_{op}(x)))] \end{aligned}$$

Nachdem ein Zusammenhang zwischen abstrakten und konkreten Vorbedingungen der Operationen hergestellt ist, müssen auch die Nachbedingungen ineinander überführbar sein, um sicherzustellen, daß die konkreten Programme die abstrakten Bedingungen erfüllen.

VI ($\forall op \in OPERATIONS$) ($\forall y \in \text{RAN}(\text{REP})) (y = \text{REP}(x) \longrightarrow$

$$(KEFFECTS_{op}(x) \longrightarrow AEFFECTS_{op}(y) \wedge KI(x)))$$

Die Äquivalenz der 'EFFECTS'-Teile wird gefordert, um auszuschließen, daß die Implementation Zustandsänderungen zuläßt, die über die spezifizierten Änderungen hinausgehen. Keramidis fordert nur die Abschwächung der konkreten Effekte. Da Zwischenzustände der konkreten Repräsentation, die während der Abarbeitung einer Implementationsoperation auftreten können, nicht sichtbar sind, können über Zwischenzustände keine Informationen fließen. Deshalb genügt die Äquivalenz der Ausführbarkeits- und 'EFFECTS'-Bedingungen zum Nachweis des korrekten Informationsflusses innerhalb der Implementationsoperationen.

Die Verifikationsbedingungen IV bis VI haben eine doppelte Bedeutung. Erstens legen sie die Vor- und Nachbedingungen der konkreten Operationen (Prozeduren) fest und geben somit den Rahmen zu deren Verifikation vor. Zweitens folgt aus der Gültigkeit obiger Bedingungen (bei festen KEFFECTS, KNBL, KPRE), daß der sequentielle Teil der Implementation die Spezifikation

korrekt realisiert. Zur korrekten Implementation der nichtse-
quentiellen und Informationsflußeigenschaften ist noch die Gül-
tigkeit der Bedingung VII zu beweisen:

$$\text{VII } (\forall y \in \text{RAN}(\text{REP}))(y = \text{REP}(x) \rightarrow (\text{ACOMP}(y) \rightarrow \text{KCOMP}(x)) \wedge \\ (\text{APRIO}(y) \rightarrow \text{KPRIIO}(x)) \wedge (\text{AINF}(y) \rightarrow \text{KINF}(x)))$$

Stellt die Implementation eines der drei Konstrukte 'INF',
'PRIO' oder 'COMP' nicht zur Verfügung, so muß aus der Semantik
der abstrakten Bedingungen und der jeweiligen Implementation
(z.B. mit Semaphoren oder dem Rendezvous-Konzept) die Kor-
rektheit eigens nachgewiesen werden. Dies gilt auch, falls die
Implementationskonstrukte semantisch nicht den Spezifika-
tionskonstrukten entsprechen. Die Gültigkeit der Bedingung (*)
aus der Bedingung (V) ergibt sich, wenn man 'PRE' und 'NBL'
konjunktiv verknüpft. Die 'ACC'-Bedingungen sind bereits darin
enthalten und brauchen in der Konjunktion nicht extra berück-
sichtigt zu werden.

Informationsflußeinschränkungen

Die korrekte Implementation der Informationsflußeinschränkun-
gen wird auf eine einfache Äquivalenz zurückgeführt. Dazu ist
zu beweisen, daß ein implementierter abstrakter Datentyp nicht
mehr Kanäle als die zugehörige Spezifikation enthalten kann.
Ansonsten könnte nur die Implikation nachgewiesen werden (siehe
5.3). Beim Vergleich von Implementation und Spezifikation fällt
auf, daß die Implementationsprogramme nach außen verborgen
sind, so daß kein Zusammenhang zwischen der Ausführungsdauer
einer Operation und dem Zustand des abstrakten Objekts herge-
stellt werden kann. Das bedeutet, daß die Implementation nicht
mehr Kanäle als die Spezifikation zur Verfügung stellt, so daß
aus der Gültigkeit der Invarianten die Äquivalenz zwischen ab-
strakten und konkreten Informationsflüssen gefolgert werden
kann. Die Gültigkeit der Informationsflußeinschränkungen folgt
dann direkt aus der Gültigkeit der Invarianten und der Semantik
des 'INF'-Konstrukts.

5.3 Korrektheit der Implementation

Durch den im zweiten Abschnitt beschriebenen engen Zusammenhang zwischen Spezifikation und Implementation und durch das 'INF'-Konstrukt der Implementationssprache IMAGAD vereinfachen sich die Informationsflußuntersuchungen der Implementation eines abstrakten Typs wesentlich. Die in Abschnitt 2 eingeführte Verifikationsbedingung (VI) garantiert, daß

- durch ändernde Operationen nur spezifizierte Änderungen am abstrakten Objekt ausgeführt werden und
- lesende Operationen nur die spezifizierten Ausgabewerte erhalten.

Daraus folgt, daß die Implementation eines abstrakten Typs nur durch Verifikation der sequentiellen Eigenschaften als korrekt bez. der angegebenen Informationsflußrelation angesehen werden kann. Voraussetzung ist, daß ein Mechanismus zur Implementation des 'INF'-Konstrukts zur Verfügung steht. Ist in der Spezifikation die Informationsflußanalyse bereits statisch durchgeführt, aufgrund einer Informationsflußeinschränkung über den Parameter 'INF_PAR', so entfällt in der Implementation auch die Angabe von entsprechenden 'INF'-Bedingungen.

Damit verbleibt in der Implementation der Nachweis über die Korrektheit der Operationen bez. der konkreten 'PRE'-, 'NBL'- und 'EFFECTS'-Bedingungen. Es ist also nachzuweisen, daß gilt:

VIII $(\forall op \in OPERATIONS)(\forall x \in DOM(REP))$

$$[(KI(x) \wedge KPRE_{op}(x) \wedge KNBL_{op}(x)) \wedge op(x) \wedge KEFFECTS_{op}(x) \wedge KI(x)]$$

Die Verifikation sequentieller Programme soll hier nicht weiter ausgeführt werden. Mit Hilfe von Zusicherungen oder dem Hoareschen Verifikationskalkül lassen sich, bei geeigneter Axiomatisierung der verwendeten Anweisungen, obige Bedingungen formal überprüfen. Zur Korrektheit gehört natürlich auch die Terminierung.

Korrekte Implementation von Informationsflußeinschränkungen:

Unter Voraussetzung der Bedingungen (IV) bis (VII) gilt für alle Variablen x der Implementation eines abstrakten Typs ADT:

$$(\forall x \in KO)(\exists op_1, op_2 \in OPERATIONS)(\inf(op_1, op_2, x, TRUE, TRUE) \rightarrow x \in DEPENDSET[op_2] \wedge x \in CHANGESSET[op_1])$$

(Es gibt also keine Variablen in der Implementation, außer dem abstrakten Objekt, über die Information mit Hilfe der Operationen fließen kann.)

Informeller Beweis:

- 1) Lokale Variablen einer Prozedur brauchen nicht berücksichtigt zu werden, da sie nicht von zwei Aktivitäten geändert bzw. gelesen werden können.
- 2) Globale Variablen eines abstrakten Typs sind nicht sichtbar.
- 3) Damit bleibt nur die Möglichkeit, daß eine Komponente des abstrakten Objekts in der Implementation anders als in der Spezifikation behandelt wird, so daß die Komponente nicht in 'CHANGESET' oder 'DEPENDSET' enthalten ist. Das würde mindestens einer der Bedingungen (IV) bis (VII) widersprechen.

5.4 Verifikation des Datentyps 'pnezev'

Der in Kapitel 3 spezifizierte und in Kapitel 4 implementierte abstrakte Datentyp 'pnezev' soll im folgenden auf Korrektheit überprüft werden. Dazu ist die Repräsentationsfunktion anzugeben, die die konkrete Repräsentation auf die abstrakte abbildet. Die abstrakten Invarianten des Datentyps dienen zur Charakterisierung der Typeigenschaften und sind im Beweis zu berücksichtigen. Der Beweis richtet sich nach der Reihenfolge der angegebenen Bedingungen (I-VIII). Zunächst werden die Invarianzeigenschaften auf Spezifikationsebene bewiesen. Anschließend ist die Korrespondenz von Spezifikation und Implementation nachzuweisen und zuletzt werden die Implementationsalgorithmen untersucht.

Verifikationsbedingungen der Spezifikation

Zur Bestimmung der abstrakten Invarianten ist zu berücksichtigen, daß für jedes Objekt des Datentyps gelten muß:

- Der Puffer darf die Länge max_elements nicht überschreiten.
(I1) $\text{LENGTH}(\text{pnezv}) \leq \text{max_elements}$
- Aus dem leeren Puffer können keine Elemente entfernt werden.
(I2) $\text{LENGTH}(\text{pnezv}) \geq 0$
- Informationsfluß über ein abstraktes Objekt vom Typ 'pnezv' mit den Operationen 'insert' und 'remove' ist verboten, wenn die 'insert'-Operation eine höhere Klassifikation als die 'remove'-Operation hat.
(I3) $(\forall \text{op}_1, \text{op}_2 \in \text{OPERATIONS})(\text{op}_1 = \text{insert}(\text{c1}) \wedge \text{op}_2 = \text{remove}(\text{c2}) \wedge \text{c1} > \text{c2} \rightarrow \text{NINF}(\text{op}_1, \text{op}_2))$

Da eine Informationsflußeinschränkung vorgegeben ist, muß entschieden werden, ob dynamische oder statische Informationsflußuntersuchungen durchgeführt werden sollen, d.h., ob in der Implementation Informationsfluß über den Parameter 'INF_PAR' erlaubt ist. Ist dies der Fall, so wird die Informationsflußbedingung nicht als Invariante aufgefaßt, sondern muß an der Schnittstelle zur Implementation verifiziert werden (Verifikationsbedingung VII). Im anderen Fall ist die Informationsflußeinschränkung als abstrakte Invariante aufzufassen, die von der Spezifikation zu erfüllen ist (Verifikationsbedingung III). Dann ist in der Implementation auch keine 'INF'-Einschränkung zu formulieren, da die Einschränkungen durch den Aufbau des abstrakten Typs statisch erfüllt sind. Im Fall des Typs 'pnezv' wird das 'INF'-Konstrukt der Implementation verwendet. Dann ist nachzuweisen, daß die formulierte 'INF'-Einschränkung der Invarianten I3 entspricht. Dies ist aus der Semantik des 'INF'-Konstrukts ersichtlich. Die Verifikation der Bedingungen I und II zeigt Abb.63.

Bedingung I verlangt, daß nach Initialisierung die Invarianten gelten.

Es ist zu zeigen: $\text{AINIT}(A0) \rightarrow I1 \wedge I2$

Aus der Spezifikation folgt, daß nach AINIT gilt:

$\text{pnezev} = \text{EMPTY_SEQUENCE}$
 $\rightarrow 0 = \text{LENGTH}(\text{pnezev}) \leq \text{max_elements}$

Bedingung II schränkt die Postbedingungen der Operationen bez. der Präbedingungen ein:

$I1 \wedge I2 \wedge \text{PRE}_{\text{op}} \wedge \text{NBL}_{\text{op}} \wedge \text{EFFECTS}_{\text{op}} \rightarrow I1[\text{pnezev}_{\text{pnezev}'}] \wedge I2[\text{pnezev}_{\text{pnezev}'}]$

Dann gilt: $\text{pnezev}' = \text{pnezev} \sim \text{elem}$ (* EFFECTS_{insert} *)
 $\wedge \text{LENGTH}(\text{pnezev}) < \text{max_elements}$ (* NBL_{insert} *)
 $\rightarrow \text{LENGTH}(\text{pnezev}') = \text{LENGTH}(\text{pnezev}) + 1$
 $\rightarrow 0 \leq \text{LENGTH}(\text{pnezev}') \leq \text{max_elements}$

Außerdem gilt: $\text{pnezev}' = \text{REMAINDER}(\text{pnezev})$ (* EFFECTS_{remove} *)
 $\wedge \text{LENGTH}(\text{pnezev}) > 0$ (* NBL_{remove} *)
 $\rightarrow \text{LENGTH}(\text{pnezev}') = \text{LENGTH}(\text{pnezev}) - 1$
 $\rightarrow 0 \leq \text{LENGTH}(\text{pnezev}') \leq \text{max_elements}$

Abb. 63: Verifikation der Bedingungen I und II

Verifikation der Schnittstelle Spezifikation - Implementation

Die Repräsentationsfunktion REP wird angegeben, indem die abstrakte Repräsentation eines 'pnezev'-Objekts durch seine konkrete Repräsentation dargestellt wird:

Mit $m_0 = \text{max_elements}$ gilt:

$\text{pnezev} = \text{elements}[\text{out} \bmod m_0] \sim \text{elements}[(\text{out}+1) \bmod m_0] \sim \dots \sim$
 $\text{elements}[(\text{in}-1) \bmod m_0]$

Die konkreten Invarianten ergeben sich direkt aus den abstrakten. Voraussetzung ist, daß 'in' und 'out' potentiell unendlich sind.

$0 \leq \text{LENGTH}(\text{pnezev}) < \rightarrow 0 \leq \text{in} - \text{out}$
 $\text{LENGTH}(\text{pnezev}) \leq \text{max_elements} < \rightarrow \text{in} - \text{out} \leq \text{max_elements}$

Die Verifikation der Bedingungen IV bis VI zeigt Bild 64.

Schließlich müssen die Synchronisations- und Informations-

Bedingung IV verlangt, daß nach Ausführung der Initialisierungsprozedur die konkreten Invarianten und die abstrakte Initialisierungsbedingung gelten.

{TRUE} in:=0; out:=0; {in-out=0 $\wedge$ 0=LENGTH(pnezev) $\leq$ max_elements}

Bedingung V stellt den Zusammenhang zwischen abstrakten und konkreten Vorbedingungen ('NBL','PRE','ACC') her.

ANBL_{insert} <---> LENGTH(pnezev)<max_elements
 <---> in-out>max_elements
 <---> KNBL_{insert}

ANBL_{remove} <---> LENGTH(pnezev)>0
 <---> in-out>0
 <---> KNBL_{remove}

Schließlich fordert Bedingung VI die Äquivalenz von konkreten und abstrakten Postbedingungen.

AEFFECTS_{insert} <---> pnezev'=pnezev-elem
 $\wedge$ LENGTH(pnezev')=LENGTH(pnezev)+1
 <---> in'-out=in-out+1 $\wedge$ elements[in MOD m₀] =elem
 <---> elements'[(in'-1) MOD m₀] =elem $\wedge$ in'=in+1
 KEFFECTS_{insert}

Analog AEFFECTS_{remove}.

Abb. 64: Verifikation der Bedingungen IV bis VI

flußbedingungen auf Äquivalenz untersucht werden. Die Syntax und die Semantik der entsprechenden konkreten Konstrukte machen die Äquivalenz offensichtlich. Zu berücksichtigen ist, daß die Bedingung KINF eingeschränkt gegenüber der Bedingung AINF gilt. Erstens können Informationsflüsse über den Parameter 'INF_PAR' nicht unterdrückt werden. Zweitens werden durch die statische Berechnung der Mengen 'DEPENDSET' und 'CHANGESET' u.U. mehr Informationsflüsse eingeschränkt, als in der Spezifikation angegeben sind. Es wird lediglich garantiert, daß nicht mehr Informationsflüsse als in der Spezifikation angegeben möglich sind.

Verifikation der Implementation

Die Verifikation geht nach der Hoareschen Verifikationsmethode vor. In den Postbedingungen sind die geänderten Komponenten der konkreten Repräsentation zu apostrophieren, um ihre Äquivalenz mit den konkreten 'EFFECTS'-Bedingungen nachzuweisen. Dazu werden Variablen, die am Ende einer Ableitungskette verschieden von ihren Anfangswerten (k_1, k_2) sind und die Komponenten der konkreten Repräsentation entsprechen, mit einem Apostroph versehen. Abb.65 zeigt die notwendigen Verifikationsschritte für die Operation 'insert'. Die Operation 'remove' ist analog zu verifizieren.

```
{in-out < max_elements ∧ in = k2 ∧ out = k3 ∧ elements[in MOD max_elements] = k1}

elements[in MOD max-elements] := elem;

{in-out+1-1 < max_elements ∧ elements[(in+1-1) MOD max_elements] = elem # k1}

INC(in);

{ (in-out-1 < max_elements <---> in-out ≤ max_elements) ∧ in # k2 ∧ out = k3
  elements[(in-1) MOD max_elements] = elem # k1 --->
  in'-out ≤ max_elements ∧ elements'[(in'-1) MOD max_elements] = elem }
```

Abb. 65: Verifikation der Bedingung VIII

6 Ergebnisse, weiterführende Arbeiten

Ergebnisse

Aus der Zusammenstellung der wichtigsten theoretischen und praktischen Methoden und Maßnahmen zur Behandlung von Schutzproblemen in Kapitel 1 wird deutlich, daß

- Schutzprobleme meist unabhängig vom eigentlichen Problem behandelt werden,
- Schutzprobleme nur mit Hilfe spezieller Mechanismen beschrieben werden,
- kein Zusammenhang zu Synchronisationsproblemen hergestellt wird,
- keine einheitliche Beschreibungsmöglichkeit für Schutzprobleme existiert.

Mit der Erweiterung der Betriebssystemmaschine um das inf-Prädikat wird eine allgemeine Beschreibungsmöglichkeit für Schutzprobleme (Informationsfluß und Zugriffsschutz) definiert. Da die Betriebssystemmaschine ein allgemeines Modell zur Beschreibung asynchroner Systeme ist, wird mit der Erweiterung die Behandlung von Schutzproblemen in umgebenden Systemen möglich. Außerdem werden Schutz- und Synchronisationsprobleme im selben Modell untersucht, so daß Wechselwirkungen einfach darstellbar sind. Kapitel 2 zeigt, daß bekannte Schutzmodelle in der erweiterten Betriebssystemmaschine simulierbar sind.

In Kapitel 3 werden abstrakte Datentypen über das Modell der Betriebssystemmaschine eingeführt und eine Sprache zu ihrer operationellen Spezifikation angegeben. Durch geeignete Konstrukte ('ACC', 'INF', 'COMP', 'PRIO') können Schutz- und Synchronisationseigenschaften eines abstrakten Typs spezifiziert werden. Damit sind Informationsfluß- und Zugriffseinschränkungen einheitlich und problembezogen (durch die umgebende Spezifikationssprache) spezifizierbar. Die Basis der Betriebssystemmaschine macht den Zusammenhang zwischen Schutz- und Synchronisation deutlich.

Die Übernahme der Synchronisations- und Schutzkonstrukte in

eine Implementationssprache vereinfacht die Implementation von Schutz- und Synchronisationsproblemen, da keine Einschränkungen spezieller Mechanismen berücksichtigt werden müssen. Die Konstrukte können mit Hilfe eines einheitlichen Laufzeitalgorithmus realisiert werden. Spezielle Mechanismen lassen sich durch 'IMPORT'-Angaben in den implementierten Einschränkungen berücksichtigen.

Wesentlichstes Ergebnis für die Verifikation von Schutzproblemen ist, daß Schutzprobleme dann korrekt implementiert sind, wenn die Informationsflußbedingungen in Spezifikation und Implementation äquivalent sind und die Implementationsprogramme die spezifizierten Anforderungen erfüllen. Wichtige Voraussetzung ist, daß keine Einzelheiten der Implementationsprogramme für aufrufende Operationen sichtbar sind. Es brauchen damit keine Informationsflußuntersuchungen in Implementationsprogrammen durchgeführt zu werden, so daß bekannte Verifikationsmethoden angewandt werden können. Die korrekte Implementation impliziert damit die Sicherheit (sicher(1), Kap.1) eines Datentyps, wenn die Sicherheit eines abstrakten Typs als die Einhaltung seiner spezifizierten Schutzeinschränkungen definiert wird.

Anwendungen der Ergebnisse

Die beschriebene Methode, die vorgestellte Spezifikations- und die vorgestellte Implementationssprache können als Kernstück einer Software-Entwicklungsmethode aufgefaßt werden, die die Entwicklung von asynchronen, geschützten Systemen unterstützt. Als Werkzeuge sind zunächst ein Spezifikationssprachenprozessor und ein Generator für Prototypen vorgesehen. Der Sprachprozessor soll eine einheitliche Syntax der Spezifikationssprache gewährleisten und für auf der Spezifikation basierende Werkzeuge eine interne Schnittstelle realisieren. Der Prototypgenerator soll aus einer Typspezifikation ein lauffähiges Programm erzeugen, das die spezifizierten Eigenschaften des Typs in Interaktion mit dem Benutzer simuliert und so eine empirische Kontrolle der Spezifikation ermöglicht. Ein Übersetzer für die Implementationssprache IMAGAD ist (ohne die Schutz-

konstrukte) in Modula-2 implementiert, der Spezifikationssprachenprozessor und der Prototypenzeuger sind in Bearbeitung.

Die dynamische Berechnung der Mengen 'DEPENDSET' und 'CHANGESET' kann den Feinheitegrad der Informationsflußkontrollen erhöhen. Dazu sind allerdings die Auswirkungen auf Synchronisationsbedingungen zu untersuchen. Der dynamische Aufwand muß durch den erhöhten Feinheitegrad gerechtfertigt sein.

Theoretische Untersuchungen

Ein abstrakter Datentyp ist über eine Interpretationsfunktion mit einer Betriebssystemmaschine verbunden. Durch eine Untersuchung der möglichen Interpretationen und ihrer Eigenschaften könnten im Modell der Betriebssystemmaschine Informationen über den jeweiligen abstrakten Datentyp gewonnen werden. Invarianzeigenschaften abstrakter Datentypen lassen sich eventuell in der zugeordneten Betriebssystemmaschine einfacher analysieren.

Ein anderer Aspekt ist die Ausnutzung der Repräsentationsfunktion zur automatischen Unterstützung der Programmierung. Durch geeignete Einschränkungen der konkreten Repräsentation könnten, analog zur Prototypenzeugung, aus der Spezifikation Implementationsprogramme zumindest halbautomatisch erzeugt werden.

Einen dritten Aspekt für weitergehende Untersuchungen bieten die Schutzkonstrukte selbst. Es fehlen Ansätze zur Analyse und zum Vergleich verschiedener Schutzmechanismen und Schutzmodelle. Das eingeführte inf-Prädikat kann möglicherweise als Referenzpunkt dienen, da die bekanntesten Zugriffsschutz- und Informationsflußmodelle damit simulierbar sind.

Schlußbemerkung

Für die Betreuung der Arbeit möchte ich mich bei Herrn Prof.Dr.H.J.Schneider und Herrn Dr.S.Keramidis sehr herzlich bedanken. Die Betrachtung der Probleme aus der Sicht der Programmiersprachen und der Betriebssysteme hatte wesentlichen Einfluß auf die Ergebnisse.

Besonderer Dank gilt auch Herrn Dr.J.Grosch für zahlreiche Hinweise und Anregungen.

Meinen Kollegen W.Behnsen, Dr.R.Gall, Dr.H.Göttler, G.Kragl, T.Küppers, Dr.M.Messerer und A.Völk danke ich für die Unterstützung während der Abfassung der Arbeit.

Anhang I: Syntax der Spezifikationssprache

Die folgende Grammatik beschreibt die Syntax der Spezifikationssprache SPAGAT. Sie verwendet die in Kapitel 3 beschriebene EBNF. Die Nomenklatur der Produktionssymbole unterscheidet sich von Kapitel 3, da dort die Beschreibung der Schutzkonstrukte im Vordergrund steht und hier die Sprache insgesamt. Deshalb sind auch die Produktionen der Schutzkonstrukte teilweise unterschiedlich. Das Nichtterminal "constant_expression" wird nicht weiter verfeinert, da es der in /WIRT80/ angegebenen Syntax folgt. Die Nichtterminale "type_name", "name" usw. werden alle zu "identifizier" abgeleitet und wurden zur Verbesserung der Lesbarkeit eingeführt.

```
specification = header
                abstract_representation_part
                [ import_part ]
                [ constant_declaration_part ]
                [ type_declaration_part ]
                [ parameter_declaration_part ]
                [ definition_part ]
                [ synchronization_part ]
                [ protection_part ]
                operation_part
                END adt_name .

header =        ABSTRACT TYPE SPECIFICATION adt_name
                [ "("generic_param_part")" ].

generic_param_part = generic_param_list { ";" generic_param_list }.

generic_param_list = name { "," name } ":" type restriction
                    | name { "," name } ":" TYPE.

restriction =   WHERE boolean_expression .

abstract_representation_part = ABSTRACT REPRESENTATION
                                representation_section { ";" representation_section }.

representation_section =
    component_name { "," component_name } ":" type
    {restriction} {INIT boolean_expression}.

import_part =    IMPORT import_name { "," import_name }.

constant_declaration_part =
    CONST constant_declaration { ";" constant_declaration }.

constant_declaration =  const_name "=" const_expression.

type_declaration_part =
    TYPES type_declaration { ";" type_declaration }.

type_declaration = type_name "=" type { restriction }.
```

```

type =          type_name
               | simple_type
               | SEQUENCE OF type
               | SET OF type
               | RECORD field_list END
               | ARRAY simple_type {"," simple_type} OF type
               | ADT import_name [ akt_generic_param_list ].

simple_type =    enumeration | subrange_type.

enumeration =   "(" identifier {"," identifier} ")".

subrange_type = "[" const_expression ".." const_expression "]".

field_list =    field_section {"," field_section}.

field_section = identifier {"," identifier} ":" type.

akt_generic_param_list = "(" generic_param_expression
                          { "," generic_param_expression } ")" .

generic_param_expression = param_expression | type_name.

parameter_declaration_part = PARAMETERS
                             parameter_declaration {"," parameter_declaration}.

parameter_declaration =
                             param_name {"," param_name} ":" type restriction.

definition_part = DEFINITIONS definition {"," definition}.

definition =      definition_name parameter_list ":" type
                  IS def_expression.

def_expression =  arithm_expression
                  | boolean_expression.

parameter_list = [ "(" param_name {"," param_name} ")" ].

```


elsif_part = { ELSIF equivalence THEN equivalence }.

negation = [NOT] boolean_primary.

boolean_primary = constant
| variable
| relation
| "(" equivalence ")"
| "(" quantor identifier ["∈" arithm_expression "]"
"(" equivalence ")"
| def_identifier akt_parameter_list.

quantor = "∀" | "∃".

relation = arithm_expression relational_operator arithm_expression
| arithm_expression interval_operator variable
interval_operator arithm_expression.

relational_operator = interval_operator
| ">" | "≥" | "=" | "≠" | "∈" | "⊂".

interval_operator = "<" | "≤".

arithm_expression = [sign] arithm_term {adding_operator arithm_term}.

adding_operator = "+" | "-" | "0" | "~" | "\".

arithm_term = arithm_factor {multiplying_operator arithm_factor}.

multiplying_operator = "*" | "/" | DIV | MOD | "∩".

arithm_factor = arithm_primary ["**" arithm_primary].

arithm_primary = constant
| variable
| def_identifier akt_parameter_list
| "(" arithm_expression ")"
| "{" identifier ["∈" arithm_expression]
"|" equivalence "}".

akt_param_list = ["(" param_expression ("," param_expression) ")"].

param_expression = equivalence | arithm_expression.

variable = identifier[""] {qualifier}.

qualifier =
 "." identifier
 | "[" arithm_expression ("," arithm_expression) "]" .

Anhang II: Zusammenstellung der Verifikationsbedingungen

Die in Kapitel 5 eingeführten und beschriebenen Verifikationsbedingungen werden der Übersichtlichkeit wegen noch einmal zusammenhängend dargestellt.

Verifikationsbedingungen der Spezifikation

- I $AINIT(A0) \longleftrightarrow AI(A0)$
- II $(\forall op \in OPERATIONS) (AI(A0) \wedge PRE_{op}(A0) \wedge NBL_{op}(A0) \wedge EFFECTS_{op}(A0, A0') \longleftrightarrow AI(A0'))$
- III $(\forall op_1, op_2 \in OPERATIONS) (\forall y \in REP((DEPENDSET[EFFECTS, op_1] \cup DEPENDSET[NBL, op_1] \cup DEPENDSET[PRE, op_1]) \cap CHANGESSET[op_2]))$
 $(inf(op_2, op_1, y, B_1, B_2) \longleftrightarrow AINF)$

Verifikationsbedingungen der Schnittstelle Spezifikation - Implementation

- IV $(\forall y \in RAN(REP)) (y = REP(x) \longleftrightarrow (KINIT(x) \longleftrightarrow AINIT(y) \wedge KI(x)))$
- V $(\forall op \in OPERATIONS)$
 $[(\forall i \in \{1, \dots, mop\}) (\forall y \in RAN(REP)) (y = REP(x) \longleftrightarrow (KI(x) \wedge ACC_{i, op}(y) \longleftrightarrow KACC_{i, op}(x)))$
 $\wedge (\forall i \in \{1, \dots, nop\}) (\forall y \in RAN(REP)) (y = REP(x) \longleftrightarrow (KI(x) \wedge PRE_{i, op}(y) \longleftrightarrow KPRE_{i, op}(x)))$
 $\wedge (\forall y \in RAN(REP)) (y = REP(x) \longleftrightarrow (KI(x) \wedge NBL_{op}(y) \longleftrightarrow KNBL_{op}(x)))]$
- VI $(\forall op \in OPERATIONS) (\forall y \in RAN(REP)) (y = REP(x) \longleftrightarrow (KEFFECTS_{op}(x) \longleftrightarrow AEFFECTS(y) \wedge KI(x)))$
- VII $(\forall y \in RAN(REP)) (y = REP(x) \longleftrightarrow (ACOMP(y) \longleftrightarrow KCOMP(x)) \wedge (APRIO(y) \longleftrightarrow KPRIOR(x)) \wedge (AINF(y) \longleftrightarrow KINF(x)))$

Verifikationsbedingung der Implementation

VIII $(\forall op \in OPERATIONS)(\forall x \in DOM(REP))$

$[\{KI(x) \wedge KPRE_{op}(x) \wedge KNBL_{op}(x)\} \rightarrow op(x) \rightarrow \{KEFFECTS_{op}(x) \wedge KI(x)\}]$

Anhang III: Literaturverzeichnis

Literatur zum Thema Schutz

- /ANDR80/ G. R. Andrews, R. P. Reitman: "An Axiomatic Approach to Information Flow in Programs", ACM Transactions on Programming Languages and Systems Vol. 2, 1 (1980), S. 56-76
- /BERS79/ T. A. Berson, G. L. Barksdale Jr.: "KSOS - Development Methodology for a Secure Operating System", Proc. AFIPS National Computer Conference Vol. 48 (1979), S. 365-371
- /CHEH81/ M. H. Cheheyl, M. Gasser, G. A. Huff, J. K. Millen: "Verifying Security", Computing Surveys Vol. 13, 3 (1981), S. 279-339
- /COHE78/ E. Cohen: "Information Transmission in Sequential Programs", in: Foundations of Secure Computation, Eds.: R. A. Demillo, D. P. Dobkin, A. K. Jones, R. J. Lipton, New York: Academic Press, 1978, S. 297-335
- /DENN76/ D. E. Denning: "A Lattice Model of Secure Information Flow", Communications of the ACM Vol. 19, 5 (1976), S. 236-243
- /DENN66/ J. B. Dennis, E. C. Van Horn: "Programming Semantics for Multiprogrammed Computations", Communications of the ACM Vol. 9, 3 (1966), S. 143-155
- /ENGL72/ D. M. England: "Operating System of System 250", International Switching Symposium, Cambridge, Mass., 1972

- /FABR74/ R. S. Fabry: "Capability Based Addressing",
Communications of the ACM Vol. 17, 7 (1974), S. 403-
-412
- /HARR76/ M. A. Harrison, W. L. Ruzzo, J. D. Ullman: "Protection
in Operating Systems", Communications of the ACM Vol.
19, 8 (1976), S. 461-471
- /JONE76/ A. K. Jones, B. H. Liskov: "A Language Extension for
Controlling Access to Shared Data", IEEE Transactions
on Software Engineering Vol. 2, 4 (1976), S. 277-285
- /JONE78/ A. K. Jones: "Protection Mechanisms and the
Enforcement of Security Policies", in: Operating
Systems, Eds.: R. Bayer, R. M. Graham, G. Seegmüller,
= Lecture Notes in Computer Science Vol. 60, Berlin:
Springer, 1978, S. 228-251
- /JONE78a/ A. K. Jones: "Protection Mechanism Models: Their
Usefulness", in: Foundations of Secure Computation,
Eds.: R. A. Demillo, D. P. Dobkin, A. K. Jones, R. J.
Lipton, New York: Academic Press, 1978, S. 237-252
- /LIPT78/ R. J. Lipton, L. Snyder: "On Synchronization and
Security", in: Foundations of Secure Computation,
Eds.: R. A. Demillo, D. P. Dobkin, A. K. Jones, R. J.
Lipton, New York: Academic Press, 1978, S. 367-385
- /MCGR79/ J. R. McGraw, G. R. Andrews: "Access Control in
Parallel Programs", IEEE Transactions on Software
Engineering Vol. 5, 1 (1979), S. 1-9
- /MERB80/ G. Merbeth: "Schutzmechanismen in Betriebssystemen",
Interner Bericht 17/80, Universität Karlsruhe, 1980

- /MOLH70/ L. M. Molho: "Hardware Aspects of Computing", AFIPS Conference Proceedings Vol. 36 (1970), S.135-141
- /MORR73/ J. H. Morris: "Protection in Programming Languages", Communications of the ACM Vol. 16, 1 (1973), S. 15-21
- /MYER80/ G. J. Myers, B. R. S. Buckingham: "A Hardware Implementation of Capability-Based Addressing", Computer Architecture News Vol. 8 (1980), S. 12-24
- /NEED77/ R. M. Needham, R. D. H. Walker: "The Cambridge CAP Computer and its Protection System", Proceedings of 6th ACM Symposium on Operating Systems Principles (Nov. 1977), in: Operating Systems Reviews Vol. 5, 11 (1977), S. 1-10
- /NEED80/ R. M. Needham: "Capabilities and Protection", GI - 10. Jahrestagung, in: Informatik Fachberichte Bd. 33, Ed.: W. Brauer, Berlin: Springer, 1980, S. 45-53
- /NEUM77/ P. G. Neumann et al.: "PSOS - a Provably Secure Operating System", Stanford Research Institute Technical Report SRI-TR-4332 (1977)
- /POPE78a/ G. J. Popek, D. A. Farber: "A Model for Verification of Data Security in Operating Systems", Communications of the ACM Vol. 21, 9 (1978), S. 737-749
- /POPE79a/ G. J. Popek, M. Kampe, C. S. Kline, A. Stoughton, M. Urban, E. J. Walton: "UCLA Secure Unix", Proc. AFIPS National Computer Conference Vol. 48 (1979), S. 355-364
- /REIT79/ R. P. Reitman: "A Mechanism for Information Control in Parallel Systems", in: Proc. 7th Symposium Operating Systems Principles (1979), S. 55-63

- /SALT74/ J. H. Saltzer: "Protection and the Control of Information Sharing in MULTICS", Communications of the ACM Vol. 17, 7 (1974), S. 388-402
- /SCHN80/ H. J. Schneider: "Wie findet ein Computer Informationen?", Umschau 80, Heft 16, 1980
- /SNYD77/ L. Snyder: "On the Synthesis and Analysis of Protection Systems", Proceedings of 6th ACM Symposium on Operating Systems Principles (Nov. 1977), in: Operating Systems Reviews Vol. 5, 11 (1977), S. 141-150
- /SNYD81/ L. Snyder: "Formal Models of Capability-Based Protection Systems", IEEE Transactions on Computers Vol. 30 (1981), S. 172-181
- /WEDE78/ H. Wedekind: "Zur Durchführung des §6,1, BDSG", Datenschutz und Datensicherung 4 (1978), S. 181-185
- /WEIS69/ C. Weissman: "Security Controls in the ADEPT-50 Time Sharing System", Proc. AFIPS Fall Joint Computer Conference Vol. 35 (1969), S. 119-133
- /WULF74/ W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack: "HYDRA: The Kernel of a Multiprocessor Operating System", Communications of the ACM Vol. 17, 7 (1974), S. 337-345

Allgemeine Literatur

- /KERAS2/ S. Keramidis: "Eine Methode zur Spezifikation und korrekten Implementierung von asynchronen Systemen", Arbeitsberichte des IMMD Bd. 15, Nr. 4, Universität Erlangen-Nürnberg, Erlangen, 1982
- /LIND76/ T. A. Linden: "Operating System Structures to Support Security and Reliable Software", Computing Surveys Vol. 8, 4 (1976), S. 409-445
- /MUCH81/ S. S. Muchnick, N. D. Jones: "Program Flow Analysis: Theory and Applications", Englewood Cliffs, Prentice-Hall, 1981
- /RITC74/ D. M. Ritchie: "The UNIX Time-Sharing System", Communications of the ACM Vol. 17, 7 (1974)
- /RUPP83/ G. Rupprecht: "Ein Konzept zur Integration von allgemeinen Sprachkonstrukten zur Synchronisation und von gemeinsamen abstrakten Datentypen mehrerer Prozesse in MODULA-2 und seine Implementierung", Diplomarbeit am IMMD 2, FAU Erlangen-Nürnberg, Erlangen, 1983
- /SCHN81/ H. J. Schneider: "Problemorientierte Programmiersprachen", Leitfäden der angewandten Informatik, Stuttgart, Teubner, 1981
- /WEDE82/ H. Wedekind: "Datenbanksysteme I", Reihe Informatik /16, Bibliographisches Institut, Mannheim, 1981
- /WILH79/ R. Wilhelm: "Attributierte Grammatiken", Informatik-Spektrum 2 (1979), S. 123-130

/WIRT80/ N. Wirth: "The Programming Language Modula-2", Report
Nr. 36, Institut für Informatik, ETH Zürich, 1980

Anhang IV: Stichwortverzeichnis

Hinweise:

1. "-" wiederholt den übergeordneten Begriff (auch mehrfach).
2. Geklammerte Seitennummern weisen auf die Definition oder eine ausführliche Beschreibung des Begriffs hin.

Abfrage	
semantische -	82, (85), 159
--menge	(87)
bedingte ---	(88)
--operation	(87)
bedingte ---	(88)
syntaktische -	(132)
ABSTRACT REPRESENTATION	(100), 170
ABSTRACT TYPE IMPLEMENTATION	(140)
Abteilung	21, 43
ACC	106, (115), 143, 175
acc-Prädikat	(90), 107, 115
Ada	27, 53, 149
ADEPT-50	23, 28, (31), 42, 92
Adressierung	31, 36, 38
Adreßmenge	(79)
Änderung	
semantische -	82, (84), 159
--smenge	(87)
bedingte ---	(88)
--soperation	(87)
bedingte ---	(88)
syntaktische -	(132)
Aktivität	(101), 114, 142, 166
Aktivitätskontrollblock	(151), 156, 165
ALGOL60	27, 52
Aliasbildung	127
Anhäufung	(18), 23
apnezev	120
Attribute	107, 113, 144
Ausführbar	(81), 102
Benutzeridentifikation	1
Benutzerklassen	42
Berechnung	(79), (82)
Bereit	(81), 102
Betriebssystem	27, 41, 136, 150
-maschine	77, (80), 104, 132, 143, 148
Blockiert	(81), 102
Blockkonzept	27
Capability	(34), 46, 55, 106, 152
-Konstrukt	56
-listen	36, 47, 68
-mechanismus	28, 34
CAP System	28, 35

CHANGESET	(153), 156, 159, 166, 172, 182
CLU	27, 53
COMP	(102), 113, 142, 150
CONCRETE REPRESENTATION	(141), 165, 170
CONST	142
Datenbank	124
Datenflußanalyse	154
Datenmenge	(79)
Datentypen	
Abstrakte -	27, 47, 53, 55, (99), 107
	131, 137, 139, 164, 172, 186
Datenschutz	1, 5, 11, 124
DEFINITIONS	142
DEPENDSET	(153), 156, 159, 166, 173, 182
dnezev	123
EFFECTS	(101), 176
EUCLID	27, 53
Falltüren	(17)
FDM	(66), 97
Grant-Konstrukt	56
Hardware	28, 31
HDM	(63), 97
HYDRA	28, 35, (46)
iAPX 286	28, 33
Identifikation	5, (9), 46, 55
Benutzer-	1
Datei-	43
Objekt-	43
Prozeß-	(109)
IMAGAD	(144), 151
Ina Jo	66
INF	106, (112), 134, 144, 155, 166, 174
INF_PAR	(148), 155, 173, 178
Inferenz	6, (10), 30
Informationsfluß	21, 28, 72, 82, 89, 109, 111, 133, (149)
allgemeiner -	(93)
bedingter -	(83)
direkter -	(89)
-analyse	67, 72, 155
-beschränkung	(114), 143, 147, 173, 177, 180
-kontrolle	(9), 14, 21, 29
-modell	10, 21, 26
-- (Denning)	(21), 44, 94
-problem	78
-relation	22
Informationsquelle/-senke	9, (86), 91, 114
Informationsträger	90
Informationsübertragung	83, (92), 94
inf-Prädikat	(89), 107, 112, 148, 155, 174
INIT_ACTIVITY	(114)

Integrität	(3), 11, 134
Invarianzeigenschaften	170, 172, 180
Kanal	
versteckter -	25, 72, 98, (132), 148, 156, 170, 173, 177
E/A-	(25)
Speicher-	(25)
Zeit-	(25)
Klammern	(32), 46
Klassenkonzept	27
Klassifikation	21, 28, 72, 119
Komponente	
abstrakte -	(100)
Konsistenzbedingungen	170
Kopieren	(17)
Korrektheit	179
kphtisch	129
Kryptographie	2, 6, (10), 15
KSOS	28
Laufend	(81), 102
Laufzeitsystem	167
Leistungsverweigerung	13
Leitungsspionage	2, 9, 30
lub-Operator	21
Markierungsbit	38, 40, 52
Maschine	
abstrakte -	63
Betriebssystem-	77, (80), 104, 132, 143, 148
Zustands-	66, 68, 79
Maschinenzustände	31
MESA	53
Modul	63, 164
Modula (-2)	27, 53, 137, 164
Modularisierung	27, 99
Monitor	57
MULTICS	32, 45
NBL	(101), 115, 142, 153, 161, 166, 175
NBL_PAR	(148), 155, 173
Nebenwirkungsfreiheit	(133), 155
nezev	103, 141
Objekt	10, 15, 18, 21, 32, 34, 70, 107
abstraktes -	(100)
Schutz-	(110), 143
Operation	(80)
ausführbare -	(80)
Schutz-	(111)
OPERATIONS	(101), 142, 153
PARAMETERS	(100)
PARin	(131)
PARout	(131)
Paßwortmechanismus	2, 5, 28, 46

phtisch	128
PLESSEY250	28, 34
pnezev	118, 160, 179
Postprozessor	167
pphtisch	130, 163
Präprozessor	165
PRE	(101), 115, 146, 153, 161, 166, 175
PRE PAR	(148), 155, 173
PRIÖ	(102), 113, 142, 150
Prioritätsrelation	(81), 102
Privatheit	(2), 11, 134
Privileg	
kleinstes -	(16)
Produktion	125
Produktionen	106
Profil	43
Programmiersprache	27, 52, 99, 136, 143, 150
PROT	(110), 144, 166
Prototyp	171
Prozeß	
-identifikation	(109)
sequentieller -	(79)
PSOS	28, 35, 51
Rechte	4, 15, 20, 31, 38, 40, 48, 68, 70
Repräsentationsfunktion	106, 121, 128, 162
rnezev	169, 181, 186
Rückruf	121
Rückstufung	(17), 111
	(25), 45
Schablonen	49
Schutz	(13), 77
-aspekte	108
-domäne	(16)
-graph	(18)
-komponenten	(110), 174
-konstrukte	137
-maßnahmen	1, 3, (27), 42, 55, 111, 136
--, logisch	(8)
--, organisatorische	4, (7)
--, physische	(8)
--, technische	4, (7)
-matrix	(15), 32, 35, 49, 52, 55
-mechanismus	(11), 26, 31, 34, 46, 51, 152
-modell	42
-objekte	(110), 143
-operationen	(111)
-parameter	(110), 145, 151
-problem	(3), 97, 140
discretionary --	(21), 120
mandatory --	(21), 120
-ringe	(32), 46
-strategie	(11), 26, 117
-umgebung	32, 48, 53
Secure UNIX	28, 59

Sicherheit	(3), (11), 134, 136
-sanforderung	64, 70
-sbedingung, einfache	64
-skern	137
-sklassen	21
-smodell	21, 64
mehrschichtiges --	64
militärisches --	21, 23, 72, 117
-sproblem	16, 20, 22, 51, 70
SIMULA67	27, 53
Software	
-entwicklung	97, 185
-system	107, 131, 139
SPAGAT	98, (106), 131, 137
-Syntax	106
SPECIAL	63
Spezifikation	51, (61), 63, 171
-smethode	51, 59, 66, 99, 107, 135
-ssprache	100
Statistik	2, 6, 15
Stern-Bedingung	64
Subjekt	10, 15, 18, 21, 32, 34, 107
SWARD	40
SYN	(102), 113, 142, 150
Synchronisation	77, 111
-seinschränkung	116, 157
-skontrollblock	140, (150)
-smaßnahmen	(9), 14, 29, 55, 136
-problem	27, 108, 128, 134, 140
System /360	31
System /38	28, 35, 38, 52
Tag-Bit	36, 38
Take-Grant-Modell	(18)
Tore	(32), 46
Trojanisches Pferd	(17)
UCLA-Kernel	68, 96
Überfliegen	(17)
UNIX	41
VAR	141
Variablenmenge	(80)
Variablenzugriff	(90), 95
VAX	164
Verbandsmodell	(21), 64, 72, 75
Verifikation	13, (14), 52, 59, 63, 98, 169, 178
Hoaresche -	56, 59, 72, 171, 178, 183
-sbedingungen	170
Verteilung	(17)
Verträglichkeitsrelation	(81), 102
Vollständigkeitspostulat	71, 98, 131
VMS	164
VM/370	28
Wartend	(81), 102

Werkzeuge	61,63
Wertfunktion	(80)
Wire-Tapping	(6)
Zugriffseinschränkung	(115),143,146
Zugriffskontrolllisten	45
Zugriffsschutz	14,16
-modell	10
-- (Harrison)	(15),95
-- Take-Grant	(18)
-problem	27,67,78,134
Zuverlässigkeit	(13),136
Z8000	28,33

Lebenslauf

Manfred Reitenspieß, Sieglitzhofer Str. 42 , 8520 Erlangen

13.12.1953

Geburt in Nürnberg als Sohn von Georg und Margarete
Reitenspieß, geb. Brechetsbauer, wohnhaft in 8504
Stein, Hirtenweg 7

September 1960 bis Juli 1965

Besuch der Volksschule in Stein-Oberweiherbuch

September 1965 bis Mai 1974

Besuch des Dürer-Gymnasiums, 8500 Nürnberg, Sielstr.
17, Schulabschluß mit Bestehen des Abiturs im Mai 1974

September 1974 bis Juli 1977

Studium der Mathematik an der Universität Erlangen-
Nürnberg, Beenden des ersten Studienabschnitts mit
dem Vordiplom im Juli 1977

September 1977 bis September 1980

Studium der Informatik an der Universität Erlangen-
Nürnberg, Beenden des Studiums mit dem Diplom im
September 1980, Thema der Diplomarbeit: Entwicklung,
Spezifikation und Implementierung eines zuverlässigen
und sicheren Betriebssystemkerns

Juni 1979

Heirat mit Gertraud Reitenspieß, geb. Wischmann

seit Oktober 1980

Wissenschaftlicher Mitarbeiter am Institut für Mathe-
matische Maschinen und Datenverarbeitung der Universi-
tät Erlangen-Nürnberg, Lehrstuhl für Programmier- und
Dialogsprachen, sowie Compiler (Prof. Dr. H.J. Schnei-
der)

In letzter Zeit sind folgende Arbeitsberichte des Instituts für
Mathematische Maschinen und Datenverarbeitung erschienen:

Band 13
=====

- Nr. 1 Schrammel, D., Bayer, W., Gliemann, H.-P., Heigl, M.,
Mertens, P.:
Ein Leistungs- und Wirtschaftlichkeitsvergleich zwischen
Klein- und Großcomputern
(Januar 1980)
- Nr. 2 Göttler, H.:
Semantische Modelle
(März 1980)
- Nr. 3 Fromm, H.J.:
Zur Modellierung der Speicherinterferenz bei hierarchisch
organisierten Multiprozessorsystemen
(April 1980)
- Nr. 4 Jäpel, D.:
Klassifikatorbezogene Merkmalsauswahl
(Juli 1980)
- Nr. 5 Bodendorf, F., Mertens, P., Haas, R., Weber, G. Wolf, G.:
SAMBA - ein Methodenbankrahmen um das Statistik-Paket SPSS
(Juli 1980)
- Nr. 6 Bartsch, B.:
Inferenz und Analyse spezieller Graphgrammatiken für die
syntaktische Mustererkennung
(September 1980)
- Nr. 7 Wurm, F.X.:
Auftragssystem für eine Mehrprozessoranlage
(Oktober 1980)
- Nr. 8 Schedelbeck, A.:
Vergleich der beiden höheren Realzeitprogrammiersprachen
ADA und PEARL.
(Oktober 1980)
- Nr. 9 Jahresbericht 1980 der Informatik.
(März 1981)

- Nr. 1 Pattern Recognition
(Research at Lehrstuhl für Informatik 5)
(Februar 1981)
- Nr. 2 Bunke, H.:
Analyse elektrischer Schaltpläne
(März 1981)
- Nr. 3 Linster, C.-U.:
SYMPOS/UNIX - Ein Betriebssystem für homogene
Polyprozessorsysteme
(Juni 1981)
- Nr. 4 Akyildiz, J.F., Bolçh, G.:
Analytic Solution Techniques for Queueing Network
Models of Computer Systems
(Juli 1981)
- Nr. 5 Lindstedt, W.G.:
Ein Verfahren zur Erstellung portabler algorithmischer
Laufzeitprogramme
(Juli 1981)
- Nr. 6 Vollmar, R.:
Zum Begriff des Parallelismus bei Polyautomaten
(Juli 1981)
- Nr. 7 Luft, A.L.:
Rationaler Sprachgebrauch und orthosprachliche
Standardisierung als Grundlagen zuverlässiger Software-
Entwicklung und -Dokumentaion
(Januar 1982)

Nr. 8 Händler, W.:

PROCEEDINGS OF A WORKSHOP ON TAXONOMY IN COMPUTER
ARCHITECTURE
(Februar 1982)

Nr. 9 Beth,Th., Hess,P., Wirl,K.:

Materialien zur Kryptographie
(März 1982)

Nr. 10 Jahresbericht 1981 der Informatik

(März 1982)

- Nr. 1 Wittmann, A.:
Ein Mechanismus für die Synchronisation paralleler
Prozesse
(Februar 1982)
- Nr. 2 Maehle, E.:
Fehlertolerantes Verhalten in Multiprozessoren -
Untersuchungen zur Diagnose und Rekonfiguration
(März 1982)
- Nr. 3 Bathelt, P.:
Vergleich von Synchronisationsmechanismen
(Mai 1982)
- Nr. 4 Keramidis, S.:
Eine Methode zur Spezifikation und korrekten
Implementierung von asynchronen Systemen
(Juni 1982)
- Nr. 5 Fromm, H.J.:
Multiprozessor-Rechenanlagen:
Programmstrukturen, Maschinenstrukturen und Zuordnungs-
probleme
(Mai 1982)
- Nr. 6 Bley, H.:
Vorverarbeitung und Segmentierung von Stromlaufplänen
unter Verwendung von Bildgraphen
(Juli 1982)
- Nr. 7 Hohl, W.:
Informatik-Sammlung
Katalog mit historischen Erläuterungen
(August 1982)
- Nr. 8 Messerer, M.:
Ein neuer Ansatz zur Parallelisierung von
Compilern
(August 1982)

- Nr. 9 Grosch, J.:
Eine Programmiersprache mit mengentheoretischen
Konstrukten und deren effiziente Implementierung
(August 1982)
- Nr.10 Kleinöder, W.:
Stochastische Bewertung von Aufgabenstrukturen
für hierarchische Mehrrechnersysteme
(August 1982)
- Nr.11 Kneißl, F.:
Realisierung von Makro-Datenflußmechanismen auf
hierarchischen Mehrrechnersystemen
(August 1982)
- Nr.12 Brendel,W.,Fan,Z.,Klar,R.,Schmielau,W.:
ERES 82
Handbuch und Fallstudie
Handbook and Case Study
(Oktober 1982)
- Nr.13 Pelz, K.:
Portable Codegenerierung in Übersetzern
für Prozess-Programmiersprachen
(Oktober 1982)
- Nr.14 Sommer,U.,Steinchen,J.:
Ein Kleinrechner-Grafiksystem zur Dokumentation
von EDV-Abläufen
(Oktober 1982)
- Nr.15 Hein, H.-W.:
Das Erlanger Spracherkennungssystem
(November 1982)

- Nr. 16 Bolch, G., Bruchner, W.:
Analytische Modelle für symmetrische Mehrprozessor-
anlagen mit dynamischen Prioritäten
(Dezember 1982)
- Nr. 17 Jahresbericht 1982 der Informatik
(März 1983)

- Nr. 1 Gall, R.:
Formale Beschreibung des inkrementellen Programmierens-
im-Großen mit Graph-Grammatiken
(März 1983)
- Nr. 2 König, R.:
Beiträge zur algebraischen Theorie der formalen Sprachen
(März 1983)
- Nr. 3 Welbourn, D.B.:
The Design of Mechanical Components and the Development
of DUCT:
17 Years of CAD/CAM in Cambridge University
(April 1983)
- Nr. 4 Woitok, R.:
Die System-Implementierungs-Sprache ALICE
(Mai 1983)
- Nr. 5 Schneider, S., Schwab, P., Renninger, W.:
Wesen, Vergleich und Stand von Software zur Produktion
von Systemen der computergestützten Unternehmensplanung
(Juni 1983)
- Nr. 6 Händler, Herzog, Hofmann, Schneider:
Projektabschlußbericht 1978 - 82 EGPA
Erlangen General Purpose Array
(Juli 1983)
- Nr. 7 Mackert, L:
Modellierung, Spezifikation und korrekte Realis-
sierung von asynchronen Systemen
(Juli 1983)

